

破解程序设计的奥秘

从算法到程序（第2版）

From Algorithm To Program

2nd Edition

徐子珊　编著

清华大学出版社

从算法到程序（第 2 版）

徐子珊　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本书第1章讨论算法设计、分析的基本概念。第2章讨论算法设计中最常用的几个数据结构，包括链表、栈、队列、二叉搜索树、散列表等。第3章讨论了算法设计的两个基本策略：渐增策略与分支策略。第1～3章的内容，为读者阅读本书以后的内容奠定了基础。第4章讨论几个代数计算的基本问题及其算法，包括矩阵运算、解线性方程组、多项式运算等。第5章讨论几个关于计算几何的基本问题及其算法，包括线段的相交判断、平面点集的凸包计算、最邻近点对问题等。第6章讨论了关于整数运算的基本问题，包括大整数的表示与运算、最大公约数计算、模运算、素数判定及整数因数分解等。第4～6章的内容为读者深入学习解决各种复杂问题奠定了解决数学计算问题的基础。第7～9章分别用回溯策略、动态规划策略及贪婪策略研究、解决计算机应用面临的最普遍、最典型的组合优化问题。第10章讨论图的搜索算法及其应用，包括深度优先搜索、拓扑排序、有向图的强连通分支计算、关节点计算、广度优先搜索、网络最大流及二部图的最大匹配等问题。第11章讨论了几个文本搜索的有趣算法，包括著名的KMP模式匹配算法、线性时间计算字符串中最长回文子串的Manacher算法、用动态规划策略寻求字符串中指定模式的最佳近似匹配的算法。对所有的的经典算法及数据结构，书中给出C语言的实现函数，形成一个通用的函数库，并详尽地加以解析。伴随各种算法的设计、分析及程序实现，书中给出了丰富多彩的应用问题及其解决方案的讨论，并给出了完整的程序代码。所有程序代码都经过反复调试，第12章介绍这些代码的使用方法。所有代码都以网络资源的方式提供给读者，访问下载地址为 www.tup.com.cn。

本书无论是对初学算法及程序设计入门的大学生读者还是对已经在职场打拼多年的程序员并有提高自身理论修养及技术水平愿望的读者都有开卷有益的意义。

图书在版编目（CIP）数据

从算法到程序/徐子珊编著. —2版. —北京：清华大学出版社，2015
ISBN 978-7-302-40076-9

Ⅰ.①从…　Ⅱ.①徐…　Ⅲ.①算法理论　Ⅳ.①O141.3

中国版本图书馆CIP数据核字（2015）第089616号

责任编辑：白立军
封面设计：
责任校对：白　蕾
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　**邮　　编**：100084
社 总 机：010-62770175　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　**印　　张**：37.75　**字　　数**：918千字
版　　次：2013年3月第1版　2015年6月第2版　**印　　次**：2015年6月第1次印刷
印　　数：1～　000
定　　价：　.00元

产品编号：061661-01

第 2 版前言

本书第 1 版已经面世近 2 年了。承蒙读者厚爱及清华大学出版社的大力支持,遂有了今天第 2 版的问世。

根据广大读者的意见反馈,在第 1 版的基础上,除对原有内容中所含明显错漏之处进行修改以外,第 2 版增加了关于文本搜索的一些有趣的算法,包括著名的 KMP 模式匹配算法、线性时间内计算给定字符串中最长回文子串的 Manacher 算法和文本串中模式最佳近似匹配的动态规划算法。所有这些算法都涵盖于第 11 章中。考虑到原来的第 11 章介绍了验证运行本书各章应用问题程序时需加载文件等细节,这对喜欢动手的读者来说是很有帮助的,所以保留了原来的内容并将第 11 章讨论的 3 个应用问题程序的运行加载信息也补充了进去,作为第 12 章。所有这些添加、改动都是为了对读者阅读本书有所帮助,并且能通过对本书的阅读能让更多的年轻朋友在信息时代具有良好的计算思维能力和操控计算机的能力。

网络已经成为人们获取信息、数据的最方便快捷的工具了。本书第 1 版中源代码是以传统的光盘形式提供给读者,本意是方便读者随手可用。第 2 版将以网络资源形式提供给读者,具体的访问地址是 www.tup.com.cn。

为使作者和读者之间更方便、直接地交流沟通,作者的 QQ 号及空间地址公布如下。

QQ 号:513410359。

空间地址:user.qzone.qq.com/513410359?ptlang=2052。

再次感谢清华大学出版社的白立军先生,没有他的支持和帮助,无论是本书的第 1 版还是今天的第 2 版都不会如此顺利地送到读者的面前。

徐子珊

2015 年 3 月

第 1 版前言

学科的基本问题和基本方法是学科方法论的基本内容。计算机能解决的仅仅是计算问题而已。什么是计算问题？有哪些典型的计算问题？如何描述计算问题是计算学科的基本问题之一，也是计算机应用的前提。将问题与数据加以形式化表示，并设计解决计算问题的算法，评价算法的运行效率是计算学科的基本方法。本书的每一章都围绕一类或一个计算问题的形式化描述和算法及其分析展开。在开卷之前，先粗线条地向读者描述一下本书。

计算问题来自现实世界，现实世界五彩缤纷，计算问题多种多样。本书按典型计算问题的分类来组织各章内容，包括计数问题、代数计算问题、计算几何问题、数论问题、组合优化问题和图的搜索问题。

计数问题是最古老的但也是人类生活须臾不能离开的计算问题，特别是在现代科技与工业领域存在大量的计数问题。用计算机快速解决计数问题是实至名归。第 1 章讨论解决计数问题的基础是加法原理和乘法原理的应用。

为了让读者清楚地看到数据组织方式对算法设计方法及算法运行效率的影响，在第 2 章中浓缩了关于线性表、二叉树、散列表等最基础的几个数据结构。

数学中的计算问题更是比比皆是。数学问题的算法，如解线性方程组、计算多项式的变换、线段之间的位置关系等也是很多信息处理应用问题中经常要用的基本操作。本书用第 4～6 章的篇幅讨论代数、几何及数论中的典型计算问题的算法，所用的方法是第 3 章中介绍的渐增性策略和分治策略。

在多个可能解中寻求最优解的组合优化问题是计算学科面对的最典型的问题，因为人类的活动几乎都涉及资源的竞争，而有资源竞争就会产生组合优化问题。本书用第 7～9 章的内容来讨论组合优化问题的解决方法。讨论是按从大到小收缩解空间的线索展开。从无约束的回溯策略开始，到加上最优子结构性质及子问题重叠性质后的动态规划策略，解空间越小，算法效率越高。笔者试图以这样的全方位的讨论组合优化问题解决方法的形式，引导读者深入理解启发式解题思想方法。

在第 10 章讨论一个描述应用问题的重要数学模型——图。重点讨论图中顶点的搜索算法。利用搜索算法，讨论了诸如拓扑排序、关节点计算、网络流等几个经典的关于图的应用问题。

算法研究历史悠久。然而今天，算法理论研究落脚点在于指导计算机程序设计实践。本书讨论的每一个经典算法，均用 C 语言写成了通用的功能函数，并用这些函数解决了一系列有趣的应用问题。第 11 章汇总了这些函数的原型声明及数据结构的定义。

本书写作上除了上述的内容组织形式上的特点外，还用心于以下几点。

1. 理论严谨，语言规范

对每一个问题的算法，从问题的分析开始，包括思路的发展，算法的描述，正确性证明，运行时间的计算都加以详尽讨论，让读者能体验到计算学科的科学严谨性。算法设计与分

析以理论繁复著称。笔者试图以朴实的文字和平和的阐述展示对问题的分析,算法步骤的思考和运行时间估算。在确保科学性、正确性的前提下尽量使用与生活语言相近的词语而避免使用过多生僻的专用术语,让读者在阅读中感受本书语言的自然亲切。

2. 理论与实践互动

笔者对每一个理论算法都给出现有技术的程序实现,用以验证理论算法的正确性。虽然此前已经在理论上证明了算法的正确性,但通过实现了的程序的正确运行进一步证明理论是可行的。并且,算法的程序实现及对测试数据的调试运行能使读者深入理解算法的思想及其中细节微妙之处。对书中的每一个经典算法,均精选了 1～2 个应用问题,或说明如何直接调用算法解决该问题,或说明如何运用算法设计的思想解决问题。问题均选自 ACM/ICPC 的赛题或北京大学的网站 http://poj.org/problemlist。

3. 小步推进,深入浅出

要设计一个算法来解决计算问题往往是比较复杂的。对复杂问题分析以及设计解决问题的算法并对其进行分析,进而实现为程序难免行文比较冗长。为减轻读者阅读疲劳,在保证内容完整性的前提下,适当地将问题分析、算法设计分析以及程序实现复杂过程按一定的内部逻辑分解成若干部分,一步一个小标题。读者可依次一步一步连续阅读,也可分多次,每次阅读一个部分。阅读时可通过小标题明确自己在整个过程中的那一部分,又不失对全局的掌控。

4. 图文并茂,生动形象

算法的基础是数学,数学讲的是逻辑思维。然而,逻辑思维并不排斥形象思维,形象思维有时可为逻辑思维深入推进助力。为帮助读者快速且正确地理解抽象概念,或思想方法,或微妙的技术细节,书中在适宜的地方插入很多精心绘制的插图。通过这些插图读者可对书中相应的文字或符号表述的理论、方法或技术内容有生动、形象的认识。

5. 通用代码,便于引用

本书中对所有算法的程序实现并非简单的代码堆砌。笔者对所实现的每一个 C 函数参数与返回值,数据与变量的设置及关键代码都进行了详尽的解析。并且将大多数算法和数据结构写成通用的代码,以光盘的形式向读者提供类似于 C++ 的 STL 或 Java 的 Collection Framework 的通用库,便于读者在工作中或生活中需要时引用。本书算法中的伪代码的编写规范参照《Introduction to Algorithm》一书中的体例。

为方便选用本书作为算法课程教材的教师朋友使用,随书光盘中提供了 PPT 格式的课件。

徐子珊　记于山城重庆

2012 年 10 月

目　　录

第 1 章　计算问题 …… 1

1.1　计算问题及其算法 …… 1

1.1.1　计算问题及其描述 …… 1

1.1.2　算法及其描述 …… 2

1.1.3　伪代码的使用约定 …… 3

1.1.4　算法分析 …… 4

1.1.5　算法运行时间的渐近表示 …… 5

1.2　数据结构 …… 6

1.2.1　什么是数据结构 …… 6

1.2.2　数据结构对算法效率的影响 …… 7

1.2.3　字典与字典操作 …… 8

1.3　程序设计 …… 10

1.3.1　算法与程序 …… 10

1.3.2　数据类型的抽象与代码通用性 …… 11

1.4　数据的输入输出 …… 13

1.4.1　应用问题 …… 13

1.4.2　标准输入输出 …… 15

1.4.3　文件输入输出 …… 20

1.5　计数问题 …… 22

1.5.1　简单模拟 …… 23

1.5.2　加法原理和乘法原理 …… 25

1.5.3　计算四边形个数 …… 31

第 2 章　数据结构基础 …… 37

2.1　线性表 …… 38

2.1.1　线性表的链表表示 …… 38

2.1.2　对链表的操作 …… 39

2.1.3　链表的程序实现 …… 42

2.1.4　链表应用 …… 47

2.2　栈 …… 53

2.2.1　栈的概念及其链表实现 …… 53

2.2.2　栈的程序实现 …… 54

2.2.3 栈的应用 …… 56
2.3 队列 …… 62
2.3.1 队列的概念及其链表实现 …… 62
2.3.2 队列的程序实现 …… 63
2.3.3 队列的应用 …… 64
2.4 二叉搜索树 …… 68
2.4.1 二叉树及其在计算机中的表示 …… 68
2.4.2 二叉搜索树 …… 76
2.4.3 二叉搜索树的查询操作 …… 76
2.4.4 二叉搜索树中元素的增删 …… 78
2.4.5 红-黑树及其性质 …… 80
2.4.6 红-黑树的操作 …… 83
2.4.7 红-黑树的程序实现 …… 92
2.4.8 二叉搜索树的应用 …… 102
2.5 散列表 …… 102
2.5.1 直接寻址表与散列表 …… 102
2.5.2 用拉链法解决冲突 …… 104
2.5.3 散列表的程序实现 …… 106
2.5.4 散列表的应用 …… 109

第 3 章 基本算法设计策略 …… 112
3.1 渐增型算法 …… 112
3.1.1 有序序列的合并问题 …… 112
3.1.2 序列的划分问题 …… 117
3.2 分治算法 …… 121
3.2.1 归并排序算法 …… 122
3.2.2 快速排序算法 …… 126
3.2.3 序统计与选择问题 …… 130
3.3 排序问题的讨论 …… 132
3.3.1 排序的性质 …… 132
3.3.2 比较型排序算法的时间复杂度 …… 133
3.3.3 应用 …… 136
3.4 堆与基于堆的优先队列 …… 141
3.4.1 堆的概念及其创建 …… 141
3.4.2 基于二叉堆的优先队列 …… 149
3.4.3 应用 …… 153

第 4 章 代数计算 …… 169
4.1 矩阵及其计算 …… 169

4.1.1 矩阵与向量…………………………………………………………………… 169
4.1.2 矩阵的运算…………………………………………………………………… 171
4.1.3 矩阵的性质…………………………………………………………………… 173
4.1.4 矩阵的程序实现……………………………………………………………… 174
4.2 矩阵的 LUP 分解 ……………………………………………………………… 176
4.2.1 LUP 分解法概述 ………………………………………………………… 177
4.2.2 LU 分解 …………………………………………………………………… 178
4.2.3 计算 LUP 分解 …………………………………………………………… 179
4.2.4 程序实现……………………………………………………………………… 182
4.3 解线性方程组 ……………………………………………………………………… 183
4.3.1 前代法和回代法……………………………………………………………… 183
4.3.2 用 LUP 分解计算矩阵的逆 ……………………………………………… 185
4.3.3 程序实现……………………………………………………………………… 186
4.4 多项式及其计算 …………………………………………………………………… 188
4.4.1 多项式及其表示……………………………………………………………… 188
4.4.2 多项式的运算………………………………………………………………… 190
4.4.3 FFT ……………………………………………………………………………… 191
4.4.4 程序实现……………………………………………………………………… 199
4.5 应用 …………………………………………………………………………………… 204
4.5.1 多项式的泰勒展开式………………………………………………………… 204
4.5.2 完善序列……………………………………………………………………… 208
4.5.3 函数的有理式逼近…………………………………………………………… 211

第 5 章 计算几何……………………………………………………………………… 218
5.1 线段的性质 ………………………………………………………………………… 218
5.1.1 叉积及其应用………………………………………………………………… 219
5.1.2 向量的极角…………………………………………………………………… 222
5.1.3 程序实现……………………………………………………………………… 223
5.2 判断是否存在线段相交 ………………………………………………………… 226
5.2.1 算法描述与分析……………………………………………………………… 227
5.2.2 程序实现……………………………………………………………………… 230
5.3 求凸壳 ………………………………………………………………………………… 234
5.3.1 Graham 扫描 ……………………………………………………………… 235
5.3.2 程序实现……………………………………………………………………… 239
5.4 求最邻近点对 ……………………………………………………………………… 242
5.4.1 算法描述与分析……………………………………………………………… 242
5.4.2 程序实现……………………………………………………………………… 245
5.5 应用 …………………………………………………………………………………… 248
5.5.1 光导管………………………………………………………………………… 248

5.5.2 最小边界矩形 …… 255
5.5.3 德克萨斯一日游 …… 260

第 6 章 数论算法 …… 264
6.1 整数的表示 …… 264
6.1.1 整数的表示 …… 264
6.1.2 整数的算术运算 …… 264
6.1.3 程序实现 …… 269
6.1.4 应用 …… 275
6.2 初等数论的概念 …… 277
6.3 最大公约数 …… 283
6.3.1 Euclid 算法 …… 284
6.3.2 EUCLID 算法的运行时间 …… 284
6.3.3 Euclid 算法的迭代版本 …… 286
6.3.4 程序实现 …… 287
6.3.5 应用 …… 289
6.4 模运算 …… 294
6.4.1 模加法和乘法 …… 295
6.4.2 解模线性方程 …… 296
6.4.3 元素的幂 …… 299
6.4.4 应用 …… 303
6.5 素数检测 …… 305
6.5.1 伪素数检测 …… 305
6.5.2 Miller-Rabin 的随机素数检测 …… 308
6.5.3 Miller-Rabin 素数检测的错误率 …… 310
6.5.4 程序实现 …… 310
6.6 整数分解 …… 313
6.6.1 Pollard 的 ρ 探索法 …… 313
6.6.2 程序实现 …… 317
6.6.3 应用 …… 320

第 7 章 回溯策略 …… 323
7.1 组合问题 …… 323
7.1.1 组合问题的例子 …… 323
7.1.2 组合问题的形式化描述 …… 325
7.2 组合问题的回溯算法 …… 326
7.2.1 解空间的树状结构 …… 326
7.2.2 解决组合问题的回溯算法 …… 328
7.2.3 回溯算法的框架 …… 333

7.3 子集树和排列树 …… 339
7.3.1 子集树问题 …… 339
7.3.2 排列树问题 …… 343
7.3.3 应用 …… 349
7.4 用回溯算法解决组合优化问题 …… 360
7.4.1 组合优化问题 …… 360
7.4.2 用回溯策略解决组合优化问题 …… 362
7.4.3 应用 …… 365

第 8 章 动态规划策略 …… 375
8.1 组装线调度问题 …… 376
8.1.1 问题描述 …… 376
8.1.2 算法设计与分析 …… 378
8.1.3 应用——牛牛玩牌 …… 381
8.2 最长公共子序列 …… 386
8.2.1 问题描述 …… 386
8.2.2 算法设计与分析 …… 386
8.2.3 程序实现 …… 389
8.2.4 应用 …… 390
8.3 0-1 背包问题 …… 398
8.3.1 问题描述 …… 398
8.3.2 算法设计与分析 …… 398
8.3.3 程序实现 …… 401
8.3.4 应用 …… 402
8.4 带权有向图中任意两点间的最短路径 …… 409
8.4.1 问题描述 …… 409
8.4.2 算法设计与分析 …… 410
8.4.3 程序实现 …… 413
8.4.4 应用——牛牛聚会 …… 415

第 9 章 贪婪策略 …… 419
9.1 活动选择问题 …… 419
9.1.1 算法描述与分析 …… 419
9.1.2 程序实现 …… 423
9.1.3 贪婪算法与动态规划 …… 424
9.1.4 应用——海岸雷达 …… 425
9.2 Huffman 编码 …… 428
9.2.1 算法描述与分析 …… 428
9.2.2 应用——*R* 叉 Huffman 树 …… 433

9.2.3 程序实现 …… 437

9.3 最小生成树 …… 443

9.3.1 算法描述与分析 …… 443

9.3.2 程序实现 …… 446

9.3.3 应用——北方通信网 …… 448

9.4 单源最短路径问题 …… 453

9.4.1 算法描述与分析 …… 453

9.4.2 程序实现 …… 456

9.4.3 应用——西气东送 …… 458

第 10 章 图的搜索算法 …… 465

10.1 深度优先搜索 …… 466

10.1.1 算法描述与分析 …… 466

10.1.2 程序实现 …… 469

10.1.3 有向无圈图的拓扑排序 …… 472

10.1.4 应用——全排序 …… 478

10.2 有向图的强连通分支 …… 482

10.2.1 算法描述与分析 …… 482

10.2.2 程序实现 …… 486

10.2.3 应用——亲情号 …… 489

10.3 无向图的双连通分支 …… 494

10.3.1 算法描述与分析 …… 494

10.3.2 程序实现 …… 497

10.3.3 应用——雌雄大盗 …… 498

10.4 广度优先搜索 …… 504

10.4.1 算法描述与分析 …… 504

10.4.2 程序实现 …… 507

10.4.3 应用——攻城掠地 …… 508

10.5 流网络与最大流问题 …… 512

10.5.1 算法描述与分析 …… 512

10.5.2 程序实现 …… 521

10.5.3 应用 …… 523

第 11 章 文本搜索 …… 528

11.1 固定模式的串匹配 …… 528

11.1.1 强力算法 …… 528

11.1.2 KMP 算法 …… 530

11.1.3 程序实现 …… 535

11.1.4 应用 …… 535

11.2　最长回文子串问题 …… 541
11.2.1　强力算法 …… 542
11.2.2　Manacher 算法 …… 543
11.2.3　程序实现 …… 547
11.2.4　应用 …… 549
11.3　近似匹配 …… 550
11.3.1　最小编辑距离 …… 550
11.3.2　最佳近似匹配 …… 552
11.3.3　程序实现 …… 555
11.3.4　应用 …… 556

第 12 章　代码实验 …… 560
12.1　头文件清单 …… 560
12.1.1　基本应用类函数 …… 560
12.1.2　数据结构类 …… 563
12.1.3　代数记算类函数 …… 566
12.1.4　计算几何类函数 …… 568
12.1.5　数论计算类函数 …… 569
12.1.6　回溯搜索类函数 …… 571
12.1.7　动态规划类函数 …… 572
12.1.8　贪婪策略类函数 …… 572
12.1.9　图的搜索类函数 …… 573
12.1.10　文本搜索类函数 …… 574
12.2　实验平台的搭建 …… 574
12.2.1　集成开发环境的安装 …… 574
12.2.2　实验项目的建立 …… 575
12.3　应用问题程序的运行实例 …… 576
12.3.1　加载程序文件 …… 576
12.3.2　调试程序 …… 578
12.3.3　各应用问题加载文件清单 …… 579
12.4　函数库的扩展 …… 587
12.4.1　向已有的源文件中添加新函数 …… 587
12.4.2　创建新的源文件 …… 588
参考文献 …… 589

第1章　计 算 问 题

1.1　计算问题及其算法

1.1.1　计算问题及其描述

众所周知，计算机并不能解决所有问题。事实上，计算机只能解决一类称为“计算问题”的问题。所谓**计算问题**，指的是问题中所涉及的事物或其属性可用数据加以表示——称为输入数据，问题的解也可以表示成数据——称为输出数据，并且可以在有限步基本计算（算术运算、逻辑运算以及数据的暂存等）后，将特定的输入数据转换成正确的输出数据的问题。解计算问题就是将输入数据转换成正确的输出数据的过程。

利用计算机解决现实问题，首先需要将问题抽象成计算问题，也就是提炼出问题的输入数据和研究问题的解的数据特性。将问题表示为输入与输出的描述。例如，在一系列数据中查找特定值元素的查找问题可以描述如下。

输入： n 个数构成的序列 $A=\langle a_1,a_2,\cdots,a_n\rangle$，特定值 x。

输出： 若序列 A 中存在元素 $A[i]$，其值等于 x，返回从左到右的第一个值为 x 的元素的下标 i。否则，返回 -1。

例如，在线性表 $A=\langle 3,6,0,4,1,7,9,5,2,8\rangle$ 中查找特定值 x 的元素。图 1-1(a)为查找值为 $x=1$ 的元素，从 $A[1]$起依次要做 5 次检测，第 1 次找到值为 1 的元素。图 1-1(b)为查找值为 $x=11$ 的元素，从 $A[1]$起依次检测完所有元素(做 10 次检测)，没有找到值为 11 的元素——最坏情形。图 1-1(c)为查找值为 $x=3$ 的元素，从 $A[1]$起仅做一次检测就找到值为 3 的元素——最好情形。

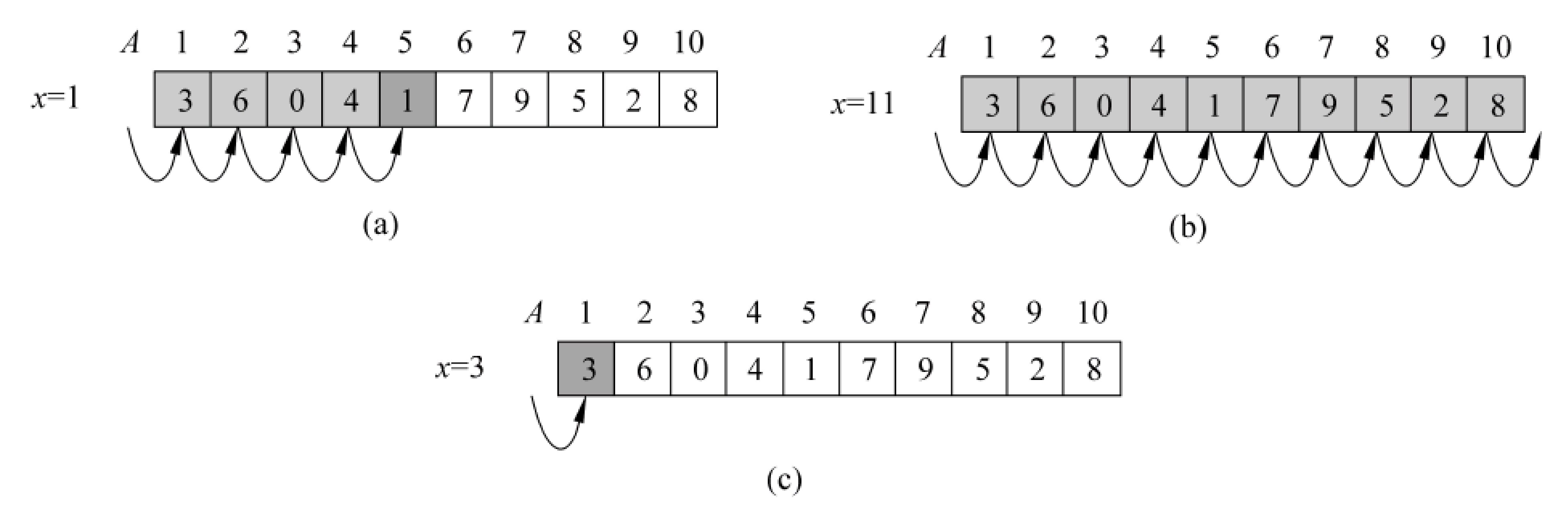


图 1-1　在一系列数据中查找特定元素

1.1.2 算法及其描述

将问题描述成其输入与输出后,就要考虑按什么样的顺序安排有限步的基本计算,将输入数据转换成输出数据,这个过程称为算法设计。安排好的计算步骤称为解决计算问题的**算法**。一个算法对问题输入的任何特定数据都能得到正确的输出数据,则称算法是正确的。对计算问题设计正确的算法,需要人们对问题有深入地理解,利用已有的数学和相关学科的科学知识以及生活常识,设计出从输入数据到输出数据的计算转换过程。例如,对上述在序列中查找特定值的问题,人们可以用如下**线性查找**算法。

解决计算问题的算法可以用各种方法加以描述。常用的有自然语言描述法、流程图描述法和伪代码描述法。用自然语言描述算法优点是表达能力很强,表述方便。例如,解决上述线性查找算法可用自然语言描述如下。

从序列的起点元素开始,依次检测元素的值是否等于 x。直至某个元素 a_i 的值等于 x,停止检测,返回 i。若检测完所有的元素(超过终点)未发现任何元素的值等于 x,则返回 -1。

用自然语言描述算法虽然方便,但有一个致命弱点:自然语言的字句可能存在歧义,即一个字句可以有不同的理解,这在描述算法时是不允许的。用**流程图**描述算法可以避免这一缺陷。仍然以线性查找算法为例,用流程图描述如图 1-2 所示。

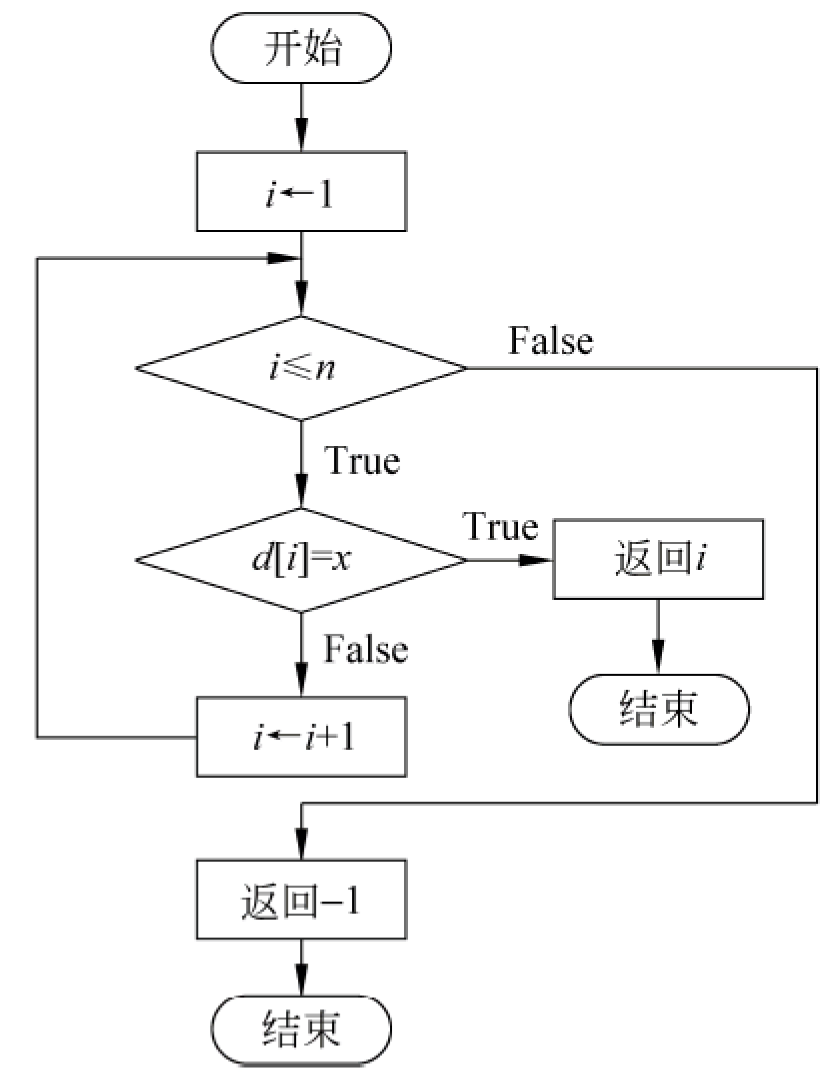


图 1-2 描述线性查找算法的流程图

用流程图描述算法克服了自然语言描述法的字句歧义性缺陷,且直观易读,但所需篇幅很大,当所要描述的算法流程比较复杂时,使用起来就不太方便。

比较实用的描述算法的工具是**伪代码**。这是一种有着类似于程序设计语言的严格外部语法(用 **if-then-else** 表示分支结构,用 **while-do** 或 **for-do** 表示循环结构),且有着内部宽松的数学语言表述方式的代码表示方法。它既没有歧义性的缺陷(严格的外部语法),又能用高度抽象的数学语言简练描述操作细节。以序列的线性查找算法为例,用伪代码描述如下。

```
LINEAR-SEARCH(A,x)
n←length[A]                  ▷n 表示序列 A 中元素个数
i←1                          ▷从 A[1]开始
while i≤n                    ▷逐一检测 A[i]的值是否等于 x
  do if A[i]=x
       then return i         ▷若存在 A[i]=x,则返回 i
     i←i+1
return −1
```

算法 1-1 在序列 A 中查找关键值为 x 的元素的线性查找过程 LINEAR-SEARCH

算法的伪代码表示除了上述的无歧义及抽象表达能力强的优势以外，有一个其他方法无可比拟的优点：其表达形式类似于高级程序设计语言代码表达形式，因此更便于将伪代码转换为程序代码。本书此后均用伪代码来描述算法。将算法描述为一个伪代码过程的其他好处在于过程的参数往往表示出了待解决的计算问题的输入，而过程的返回值表示出了该计算问题的输出。例如，解决在序列 A 中查找关键值为 x 的元素的算法 LINEAR-SEARCH(A,x)过程的参数 A 和 x 恰为该问题的输入，而在过程中第 6 行返回的 i 或在第 7 行返回的 -1 恰为问题的输出。

1.1.3　伪代码的使用约定

(1) 用分层缩进来指示块结构。例如，从第 2 行开始的 **while** 循环的循环体由 3 行组成，分层缩进风格也应用于 **if-then-else** 语句，如第 4～5 行的 **if-then** 语句。使用分层缩进而不是传统的诸如 **begin** 和 **end** 语句来表示块结构，大大降低了混乱，提高了清晰度。

(2) 循环结构 **while**、**for** 和 **repeat** 以及条件结构 **if**、**then** 和 **else** 具有与 Pascal 相仿的解释。然而，对 **for** 循环却有一点点不同：在 Pascal 中，循环计数变量的值在退出循环后是没有定义的，但在本书里，循环计数器在退出循环后仍然保留。所以，一个 **for** 循环刚结束时，循环计数器的值首次超过 **for** 循环上界。在插入排序正确性的讨论中就用到了这一特性。**for** 循环开头是 **for** $j \leftarrow 2$ **to** $length[A]$，所以当此循环结束时，$j = length[A]+1$（或等价地，$j=n+1$，因为 $n=length[A]$）。

(3) 符号 ▷ 表示本行其余部分是注释。

(4) 多重赋值形式 $i \leftarrow j \leftarrow e$ 的含义是变量 i 和 j 同赋予表达式 e 的值，它应当被理解为在赋值操作 $j \leftarrow e$ 之后紧接着赋值操作 $i \leftarrow j$。

(5) 变量(如 i、j 及 key)都局部于给定的过程。人们将不使用全局变量除非特别声明。

(6) 数组元素是通过数组名后跟括在方括号内的下标来访问。例如，$A[i]$表示数组 A 的第 i 个元素。记号“..”用来表示数组中取值的范围。因此，$A[1..j]$表示 A 由$A[1]$、$A[2]$、…，$A[j]$ j个元素构成的子序列。

(7) 组合数据通常组织在**对象**中，其中组合了若干个**属性**或**域**。用域名紧跟包括在方括号中的对象名来访问一个具体的域。例如，把一个数组当成一个对象，它具有说明其所包含的元素个数的属性 $length$。$length[A]$表示数组 A 的元素个数。虽然对数组元素和对象属性都使用方括号，通过上下文应当是能清楚辨别的。

表示数组或对象的变量被当成一个指向表示数组或对象的指针。对一个对象 x 的所有域 f，设 $y \leftarrow x$ 将导致 $f[y]=f[x]$。此外，若设 $f[x] \leftarrow 3$，则不仅有 $f[x]=3$，且有 $f[y]=3$。换句话说，赋值 $y \leftarrow x$ 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象，此时，给它一个特殊的值 NIL。

(8) 过程的参数是**按值**传递的。被调用的过程以复制的方式接受参数，若对参数赋值，则主调过程不能看到这一变化。若传递的是一个对象，则指向数据的指针被复制，但对象的域没有复制。例如，若 x 是一个被调过程的参数，过程中的赋值 $x \leftarrow y$ 对主调过程是不可见的。但是，赋值 $f[x] \leftarrow 3$ 却是可见的。

(9) 布尔运算符 and 和 or 都是**短回路**的。也就是说，当人们计算表达式 x and y 时，先

计算 x。若 x 为 FALSE,则整个表达式不可能为 TRUE,所以人们不再计算 y。另一方面,若 x 为 TRUE,必须计算 y 以确定整个表达式的值。类似地,在表达式 x or y 中,计算表达式 y 当且仅当 x 为 FALSE。短回路操作符使得人们能够写出诸如"$x \neq$NIL and $f[x]=y$"这样的布尔表达式而不必担心当 x 为 NIL 时去计算 $f[x]$。

1.1.4 算法分析

解决同一个问题,算法不必是唯一的。对表示问题的数据的不同的组织方式(数据结构),解决问题不同的策略(算法思想)将导致不同的算法。例如,若查找问题描述如下。

输入: n 个数构成的序列 $A=\langle a_1, a_2, \cdots, a_n \rangle$,其中 $a_i \leqslant a_{i+1}$, $1 \leqslant i < n$;特定值 x。

输出: 若序列 A 中存在元素,其值等于 x,返回一个值为 x 的元素在序列 A 中的位置——下标。否则,返回 -1。

可以用如下的算法来解决这一问题。

```
BINARY-SEARCH(A, x)
p←1, r←length[A]
while p≤r
  do q←⌊(p+r)/2⌋
    if A[q]=x
      then return q
    if A[q]<x
      then p←q+1
      else r←q−1
return −1
```

算法 1-2 在有序序列 A 中查找关键值为 x 的元素的二分查找过程 BINARY-SEARCH

该算法利用序列 A 的有序性,从 $p=1$, $r=n$ 开始,考察 A 的中点处元素 $A[\lfloor (p+r)/2 \rfloor]$的值是否等于 x,若是,则 q 就是所要求的下标值。否则,若该值小于 x,则丢弃 $A[p..q]$,在 $A[q+1..r]$中查找(调整 p 为 $q+1$),若该值大于 x,则丢弃 $A[q..r]$,在 $A[p..q-1]$中继续查找(调整 r 为 $q-1$)。周而复始,直至 $p>r$,即在 A 中无元素的值等于 x,返回 -1。

解决同一问题的不同的算法,消耗的时间和空间资源可能有所不同。算法运行所需要的计算机资源的量称为算法的复杂性。一般来说,解决同一问题的算法,需要的资源量越少,人们认为越优秀。计算算法运行所需资源量的过程称为算法复杂性分析,简称为**算法分析**。理论上,算法分析既要计算算法的时间复杂性,也要计算它的空间复杂性。然而,算法的运行时间都是消耗在已存储的数据处理上的,从这个意义上说,算法的空间复杂性不会超过时间复杂性。出于这个原因,人们多关注于算法的时间复杂性分析。本书中除非特别说明,所说的算法分析,局限于对算法的时间复杂性分析。

为客观、科学地评估算法的时间复杂性,人们设置一台抽象的计算机。它只用一个处理机,却有无限量的随机存储器。它的有限个基本操作——算术运算、逻辑运算和数据的移动(比如对变量的赋值)均在有限固定时间内完成,人们进一步假定所有这些基本操作都消耗一个时间单位,称此抽象计算机为随机访问计算机,简记为 RAM(Random Access

Machine)。算法在 RAM 上运行时所需的时间,显然就是执行基本操作的次数。不难看出,一个算法的时间复杂性与输入的规模相关,一般来说,规模越大,需要执行的基本操作就越多,当然运行时间就越长。例如,在对列表的搜索问题的算法中,列表所含元素个数(输入规模)越大,所花费的时间就越多。此外,即使问题输入的规模一定,不同的输入,也会导致运行时间的不同。很多文献对一个算法的运行时间,研究如下 3 种情形。

(1) 对固定的输入规模,使得运算时间最长的输入所消耗的运行时间称为算法的**最坏情形时间**。

(2) 对固定的输入规模,使得运行时间最短的输入所消耗的时间称为**最好情形时间**。

(3) 假定对固定的输入规模 n,所有不同输入构成的集合为 D_n,对问题的每一个输入 $I\in D_n$,若已知该输入发生的概率为 $P(I)$,对应的运行时间为 $T(I)$,运行时间的数学期望值 $\sum\limits_{I\in D_n} P(I)T(I)$ 称为算法的**平均情形时间**。

以线性查找算法 1-1 为例(见图 1-1),这个算法对于无解输入(即输入的线性表 $A[1..n]$ 中不存在值等于 x 的元素),所消耗的时间最长,因为第 3～4 行的 **while** 循环需要重复检测 n 次(也就是要做 n 次逻辑运算)。于是,最坏情形时间为 n。

当输入的线性表中第一个元素 $A[1]$的值就等于 x,则算法仅做一次检测就可返回答案。这是最好的情形,所以该算法的最好情形时间为 1。

假定第一个值等于 x 的元素等概率地分布在 $A[1..n]$中,也就是说第一个等于 x 的元素 $A[i]$的下标 i 为 $1,2,\cdots,n$ 的概率均为 $1/n$。这样,第 3～4 行的 **while** 循环要做 i 次检测的概率为 $1/n$。所以算法的平均情形时间为

$$\sum_{i=1}^{n} P(\text{做 } i \text{ 次检测}) \times i = \sum_{i=1}^{n} \frac{1}{n} \times i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

显然,算法的最好情形时间是有“欺骗性”的,而平均情形时间的研究要用到概率统计的知识。算法最坏情形时间可视为算法对固定输入规模 n 的运行时间的上界,用它来表示算法的时间复杂性是合理的。本书若无特殊说明,就将算法的最坏情形时间称为算法的**运行时间**。把算法的运行时间记为 T,输入的规模记为 n。则根据以上说明知 T 是 n 的正值递增函数,以 $T(n)$来表示算法的运行时间。

对于算法 1-2 中描述的二分查找而言,最坏情形发生于第 2～8 行的 **while** 循环没有发生第 5 行的 **return** 命令的执行。这样该循环的退出是由于循环条件 $p\leqslant r$ 的不满足导致的。注意,循环的每次 p 或者 r 的值都发生改变,使得 p、q 间的间距都是原来两者间距的一半。所以,循环的次数为 $\lg n$。此例说明,解决问题的不同算法,运行时间可能是不同的。

1.1.5　算法运行时间的渐近表示

由于计算机技术不断地扩张其应用领域,所要解决的问题输入规模也越来越大,所以对固定的 n 来计算 $T(n)$的意义并不大,人们更倾向于评估当 $n\rightarrow\infty$时,$T(n)$趋于无穷大的快慢来分析算法的时间复杂性。人们往往用几个定义在自然数集 **N** 上的正值函数 $\tilde{Y}(n)$:幂函数 n^k(k 为正整数)、对数幂函数 $\lg^k n$(k 为正整数,在算法研究领域,一般将常用对数 $\lg x$ 的底数设置为 2。本书若无特别说明,均按底为 2 表示常用对数)和指数函数 a^n(a 为大于 1

的常数)作为“标准”,研究极限:

$$\lim_{n\to\infty}\frac{T(n)}{\widetilde{Y}(n)}=\lambda \tag{1-1}$$

若λ为正常数,人们称$\widetilde{Y}(n)$是$T(n)$的渐近表达式,或称$T(n)$**渐近等于**$\widetilde{Y}(n)$,记为$T(n)=\Theta(\widetilde{Y}(n))$,这个记号称为算法运行时间的渐近$\Theta_-$记号,简称为$\Theta_-$记号。例如,$T(n)=3n^2+2n+1$,由于

$$\lim_{n\to\infty}\frac{T(n)}{n^2}=\lim_{n\to n}\frac{3n^2+2n+1}{n^2}=3\neq 0$$

所以,有$T(n)=\Theta(n^2)$,即此$T(n)$渐近等于n^2。其实,在一个算法的运行时间$T(n)$中省略最高次项以外的所有项,且忽略最高次项的常数系数,就可得到它的渐近表达式$\Theta(\widetilde{Y}(n))$。用此方法也能得到$3n^2+2n+1=\Theta(n^2)$。

如果两个算法的运行时间的渐近表达式相同,则将它们视为具有相同的时间复杂度的算法。显然,渐近时间为对数幂的算法优于渐近时间为幂函数的算法,而渐近时间为幂函数的算法则优于渐近时间为指数函数的算法。人们把渐近时间为幂函数的算法称为具有多项式时间的算法。渐近时间不超过多项式的算法称为**有效的**算法。本书讨论的大多数问题都有解决它的有效算法,第5章将讨论一些至今无法知道其是否有“有效的”算法的问题,这些问题导致算法研究的核心问题,也是计算机科学的核心问题——NP难问题。

渐近记号除了Θ外还有两个常用的O记号和Ω记号。它们的粗略意义如下。

考察定义域为自然数集**N**的正值函数$\widetilde{Y}(n)$和$T(n)$构成的极限式1-1的值λ,若$\lambda\geqslant 0$且λ为一常数,则称函数$T(n)$渐近不超过函数$\widetilde{Y}(n)$,记为$T(n)=O(\widetilde{Y}(n))$;若$\lambda>1$且λ为常数或为$+\infty$,则称函数$T(n)$渐近不小于函数$\widetilde{Y}(n)$,记为$T(n)=\Omega(\widetilde{Y}(n))$。例如$\lg^k n=O(n^k)$,反之,$n^k=\Omega(\lg^k n)$。显然,$T(n)=\Theta(\widetilde{Y}(n))$当且仅当$T(n)=O(\widetilde{Y}(n))$且$T(n)=\Omega(\widetilde{Y}(n))$。

1.2 数据结构

1.2.1 什么是数据结构

人们已经知道计算问题有明确的输入和输出数据,算法就是将问题的输入数据转换成正确的输出数据的操作步骤序列。在数学中,集合是表示数据的最基本的形式。相应地,在计算科学中,人们总可以把计算问题的输入输出数据表示成集合。于是,如何在计算机中表示集合就成了一个很重要的问题。在计算机中表示一个集合有两个方面的意义:其一,集合中的元素之间有什么样的关系;其二,在计算机的内部存储器中如何存储集合中的各个元素。关于集合表示的这两个方面问题的研究催生出了一门重要的课程——**数据结构**。简单地说,数据结构就是研究如何在计算机中表示一个集合,该集合中的元素间有着某种特殊关系。

例如,由n个元素构成的集合**A**,将元素排成一列$\{a_1,a_2,\cdots,a_n\}$。其中,元素a_1没有前驱元素,a_n没有后继元素,其他的元素均有一个前驱元素和一个后继元素,这样的集合称为一个**线性表**。把集合的元素按某种关系进行组织,称为数据的**逻辑结构**,例如,线性表就是一种逻辑数据结构。

集合的逻辑结构除线性表外，还可以组织成一棵**二叉树**：或者为空集，否则集合中所有元素中只有一个元素没有父结点——称为**根**结点，其他的元素（有的话）有且仅有一个父结点。包括根结点在内，每个结点至多有两个孩子结点。没有孩子的结点称为**树叶**结点，至少有一个孩子的结点称为**内**结点。内结点的孩子分成左孩子和右孩子。例如，图 1-3 表示组织成一棵二叉树的集合{A，B，C，D，E，F}。其中，A 是根结点，D、E、F 是叶子结点，A、B、C 都是内结点。B、C 分别是 A 的左、右孩子结点。B 的左右孩子结点分别是 D 和 E，C 只有左孩子结点 F。

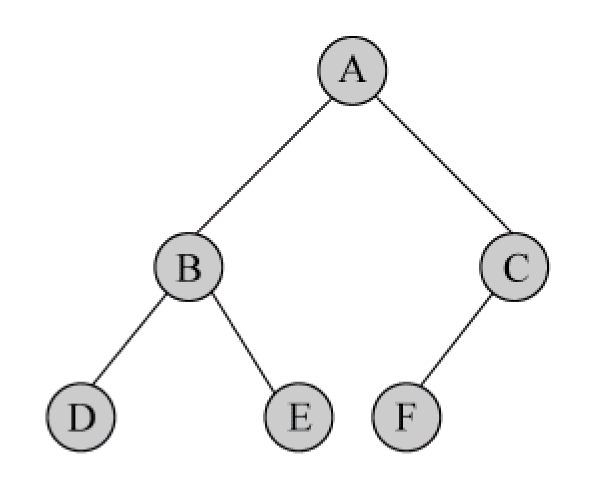


图 1-3　集合{A，B，C，D，E，F}组织成一棵二叉树

可以利用集合的线性表结构和二叉树结构组合构造出更复杂的集合逻辑结构，例如，将在以后的章节中看到的**图**的邻接矩阵结构和邻接表结构就是线性表的组合。

具有逻辑结构的集合在内存中的存储方式，称为数据的存储结构。集合的存储结构通常分为**连续存储结构**和**离散存储结构**两种。连续存储指的是将内存中的一片空间平均分成若干彼此相邻的存储单元，每个单元存放一个元素。很多高级程序设计语言（例如 C 语言）为程序员提供数组，这种数据类型就可以用来实现集合的连续存储结构。例如，用数组表示线性表，元素的下标刚好表示出线性表中元素间的前驱-后继关系，如图 1-4(a)所示。

离散存储结构指的是集合中的元素离散地存储于内存的各处，利用指针链接方式将分散存储的各元素按其逻辑关系连接在一起。例如，可以将线性表表示成链表，如图 1-4(b)所示。同样地，表示成二叉树的集合的存储结构也可以借助数组表示为连续的或借助于指针链接表示成离散的。这些正是本书第 2 章要展开讨论的话题。

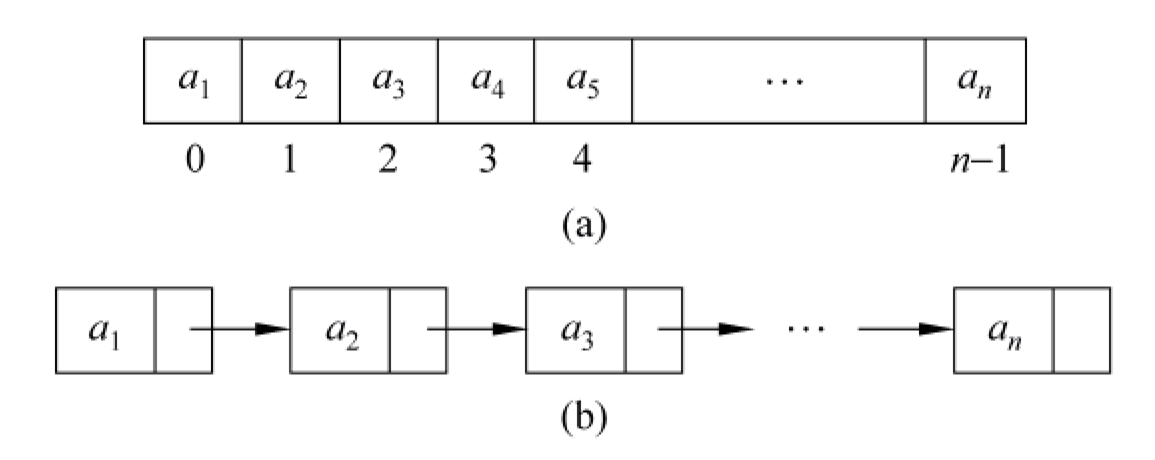


图 1-4　线性表的两种存储结构

1.2.2　数据结构对算法效率的影响

数据组织成不同的结构，是为了适应解决不同的问题，便于算法对数据的操作，提高算法的执行效率。例如，若解决一个问题的算法中要频繁地随机访问线性表中的各个元素，则线性表表示成数组比表示成链表更方便，这是因为数组中的元素可以按其下标方便地被随机访问，而在链表中访问任何结点都必须从表首结点开始顺序访问到指定结点。另一方面，如果算法需要在线性表中频繁地对元素做插入、删除操作，则将线性表表示成链表比较合适，这是因为在数组中插入或删除一个元素都可能引起大量元素的移动操作，而在链表中插入或删除一个结点只需要对个别结点的链域做改写操作。

数据结构的不同选择会影响算法的运行效率,另一个例子是考虑在具有 n 个元素的集合 A 中查找特定的值为 x 的元素问题。若将集合 **A** 表示为线性表,则人们看到算法 1-1 中的 LINEAR-SEARCH 过程的运行时间 $T(n)=\Theta(n)$。而将 **A** 表示成一个有序的线性表,则算法 1-2 中的 BINARY-SEARCH 过程将在 $\Theta(\lg n)$时间内运行完成。表面上看,算法 1-2 的运行效率比算法 1-2 的运行效率更高。然而,别忘了要将集合 **A** 表示成有序的线性表是需要花时间的。在第 3 章里读者将看到任何一种对线性表的排序算法至少要用 Θ(n)的时间。

另一方面,如果把集合 **A** 表示成一棵二叉搜索树,二叉树中根结点的值大于左孩子中任何结点的值,同时不超过右孩子中的所有结点的值,如图 1-5 所示。在本书第 2 章中读者会看到在这棵树中查找特定值 x 的结点所用的时间为 $\Theta(h)$,其中 h 为数的高度。当 n 个结点的二叉搜索树为平衡树时,树高 h 为 $\lg n$。图 1-6 展示了集合 **A**={15,18,20,6,3,7,17,13,2,9,4}表示成如图 1-5 这样二叉搜索树后在其中查找关键值 $x=9$ 的元素的操作。在图中可看到,为完成查找,x 仅与 4 个结点的数据进行了比较。而若将集合 **A** 表示为数组,则完成同样的工作需要做 10 次比较。更进一步说,若能将集合表示成一棵平衡二叉搜索树(直观地看,就是根的左右子树高度相当),在 **A** 中查找特定值为 x 的结点真的可以在$\Theta(\lg n)$时间内完成。

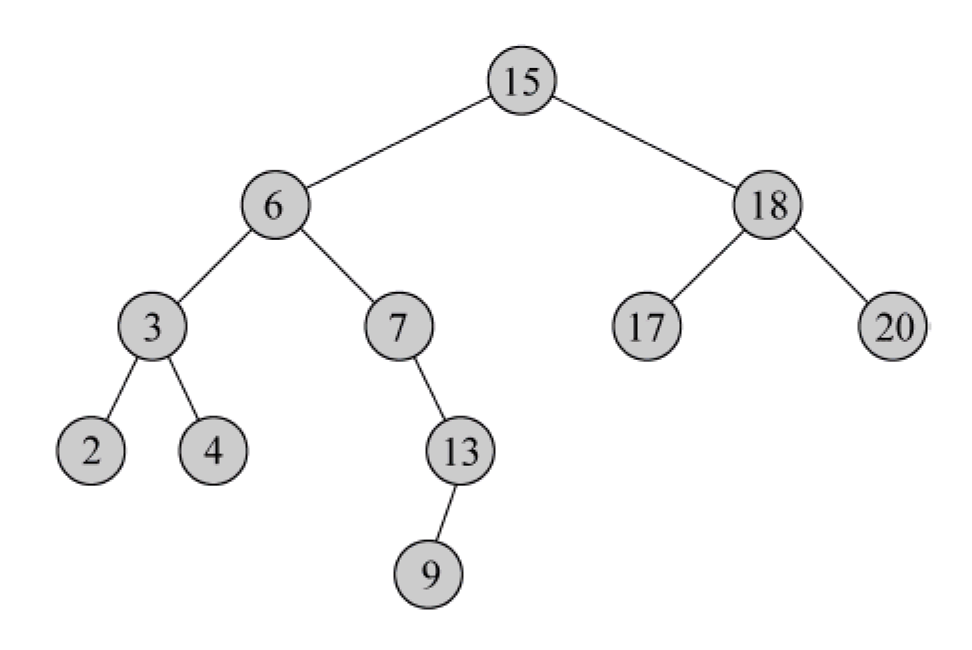


图 1-5　将集合{15,18,20,6,3,7,17,13,2,9,4}组织成一棵二叉搜索树

图 1-6 所示为在二叉搜索树中查找。图 1-6(a)中,集合 **A**={15,18,20,6,3,7,17,13,2,9,4}表示成一棵二叉搜索树,关键值 $x=9$。图 1-6(b)中,x 与根 15 比较,小于根,将在左子树中继续查找。图 1-6(c)中,x 比树根 6 大,将继续在右子树中查找。图 1-6(d)中,x 比树根 7 大,将继续在右子树中查找。图 1-6(e)中,x 小于树根 13,将继续在左子树中查找。图 1-6(f)中,找到关键值为 9 的元素。

1.2.3　字典与字典操作

如 1.2.2 节所述,研究数据集合的逻辑结构和存储结构的主要目标就是如何表示计算问题的输入输出数据,使得解决该问题的算法效率更高(运行得更快)。人们在长期的算法研究中发现,很多算法都是以下列的对数据集合的操作作为对数据的基本处理的。

设 **S** 为具有 n 个数据元素的集合。

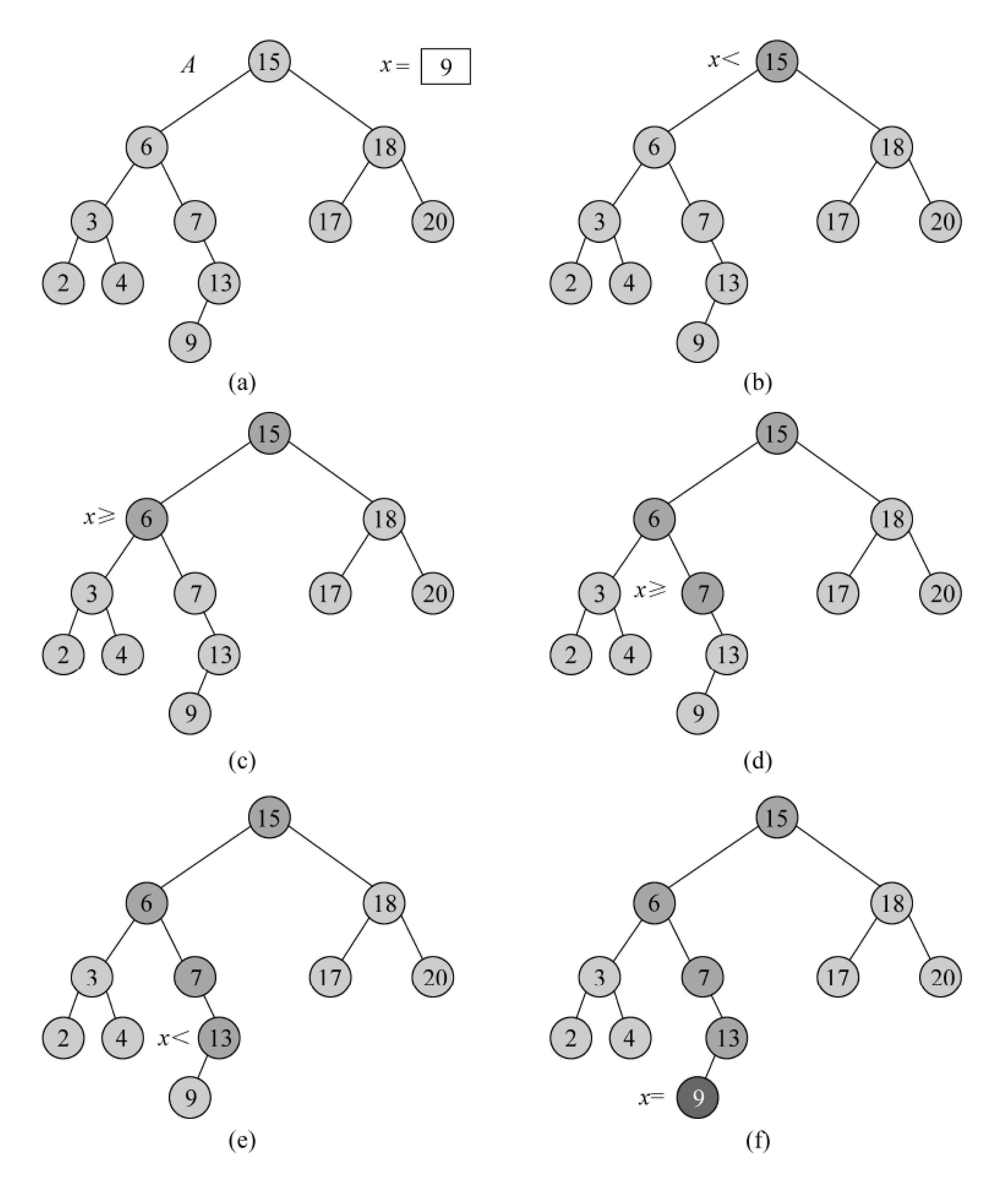


图 1-6　在二叉搜索树中查找

INSERT(S,x),在集合 **S** 中插入关键值为 x 的数据元素,即 $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{x\}$。

SEARCH(S,x),在集合 **S** 中查找关键值为 x 的数据元素,即判断是否 $x \in \boldsymbol{S}$。

DELETE(S,x),在集合 **S** 中删掉元素 x,即 $\boldsymbol{S} \leftarrow \boldsymbol{S} - \{x\}$。

上述 3 个对集合的操作称为**字典**操作,实现了字典操作的集合简称为字典。字典在很多应用问题中都是重要的数据集合表示。例如,个人通讯录、仓库物品清单、单位员工信息表等。一个算法中往往以字典操作作为基础操作,组合成更复杂的操作,来获得计算结果。本书第 2 章将讨论几个典型的字典:线性表、二叉搜索树、散列表等。除此之外,第 2 章中还将讨论插入、删除操作限制在线性表的某一端的数据结构——栈和队列,以及便于访问全序集合中最大/最小元素的数据结构——优先队列等。本书后面各章将运用这些数据结构设计解决各种计算问题的算法,并用 C 程序设计语言将其实现为可运行的程序。此外,在必要的时候,还会介绍其他有用的数据结构。

1.3 程序设计

1.3.1 算法与程序

用计算机解决计算问题,在理解了问题本身,选择好了合适方式表示出输入输出数据,并设计好了正确的算法后,接下来就要考虑如何将算法实现为计算机能够运行的程序。计算机程序就是用一种计算机程序设计语言,将算法表示成计算机能执行的数据操作指令序列。计算机技术发展到今天,曾经投入使用的程序设计语言不下千种,即使是目前流行的程序设计语言至少也有数十种,这些语言有着各自的技术特色,适应于各种类型的应用。用什么语言来实现算法,并没有一个明确的答案,这取决于算法所解决的问题性质、实现算法所需要的技术特点以及程序员对语言的熟悉程度。C 语言是当今软件开发领域最热门的语言,掌握 C 语言几乎成了跨入程序员门槛的必经之路。本书即以 C 语言为工具,深入讨论算法的程序实现,向读者展示它的魅力,从中领悟程序设计的规律,体验程序设计成功的愉悦。

作为例子,我们用 C 语言来实现在表示为数组的线性表 A 中查找特定关键值 x 元素的算法 LINEAR-SEARCH(A,x)算法过程。在 C 语言中,往往把实现某一功能的子程序写成一个函数。

```
int linearSearch(int * a,int n,int x){
    int i=0;
    while(i<n){                /* 在 a[0..n-1]中扫描 */
        if(a[i]==x)            /* 找到关键值为 x 的元素,其下标为 i */
            return i;
        i++;
    }
    return -1;                 /* 未找到关键值为 x 的元素 */
}
```

程序 1-1 实现算法 1-1 的 C 源代码文件 search.c

尽管程序 1-1 中的 C 代码与算法 1-1 中的伪代码十分相似,但要强调的是用伪代码描述的算法并不等同于程序。这主要出于如下几个原因。

(1) 算法的伪代码描述着眼于算法思想的简明阐述,高度抽象是它的最基本的特征之一。例如,在伪代码中可以用求和符号 $\sum_{i=1}^{n} x_i$ 简约地表示序列 $\langle x_1, x_2, \cdots, x_n \rangle$(尖括号括起来的元素集合强调元素的位置顺序,花括号括起来的元素集合表示一般的元素集合)的累加;而在程序中,也许就要用一个循环结构来表示这个累加过程。

(2) 算法的伪代码描述有时并不关心数据的存储格式。例如,在算法 1-1 中无须说明序列 $\langle a_1, a_2, \cdots, a_n \rangle$ 是存储在一个数组中还是存储在一个链表中,这在程序中却是必须明确声明的。

(3) 算法伪代码描述对变量无须事先声明,只要在上下文中能识别各变量及其用途就可以了;而在选用的 C 语言中所有的变量都必须先定义后使用。

可见,算法伪代码描述是写给人看的,它是人们用来设计程序的蓝图;而实现算法的程序是按设计蓝图施工而得到的成品,是写出来让机器运行的。

将算法的伪代码过程实现为程序中的函数需要仔细考虑如下 3 个要点。

(1) **参数与返回值**。伪代码过程的参数反映了待解决的计算问题的输入,而过程可能的返回值,表示该问题的输出。对于实现算法过程的函数而言,必须考虑要用多少个参数表示算法过程的各个参数(程序中可能需要多个参数表示算法中的一个参数)。它们各自是什么类型的?需要返回多少个数据,这些数据类型如何?例如,实现算法 1-1 的程序 1-1,函数名与算法的过程名相同,参数 **int** *a 和 **int** n 表示算法过程的参数序列 *A*。参数 int x 表示算法过程的参数 x。这实际上是将序列 *A* 限制为由整型元素构成用数组存储(**int** 型指针 *a 表示数组的首元素地址,参数 **int** n 表示存储于该数组的元素个数,此即为需要多个参数表示算法过程的一个参数的一个实例)的线性表。当然待查找的关键值 x 也限制为整型数据了。至于函数的返回值与算法过程是保持一致的,都是整型值(第 5 行的 **retun** i 和第 8 行的 **return** −1 对应算法 1-1 中的第 5 行和第 7 行),因此该函数的返回值类型为 **int**。

(2) **数据设置**。由于 C 语言中对所有的变量必须先定义后使用,所以要考察算法过程中所需要访问的所有数据变量,对每个变量考虑它的数据类型、存储类型、访问限制和初始值等。例如,在算法 1-1 中需要变量 i 表示扫描序列元素时的下标,所以程序 1-1 的第 2 行定义了对应的整型变量 i 且初始化为 0。

(3) **关键代码**。前面谈到,伪代码的表述可以是很简约的,其抽象程度可能是目前的程序设计语言所不能企及的,这就需要用程序设计语言提供的技术来为计算机将这些伪代码的抽象描述解释为计算机可理解的语句或表达式。此外,伪代码与程序代码之间有着一些微妙的区别。

① 伪代码中序列的下标往往从 1 开始编号,而 C 语言中,数组的元素下标是从 0 开始编号的。

② 伪代码中变量的赋值符号←明确地表示出了赋值操作的方向性,而在 C 语言中使用运算符=表示赋值运算。

③ 在 C 语言中有一套极具特色的运算符,即自增/自减运算符。用 i++(或++i)表示i←i+1,i−−(或−−i)表示 i←i−1。

所有这些,都必须在实现代码中一一体现。

1.3.2　数据类型的抽象与代码通用性

其实,程序 1-1 并不是对算法 1-1 的彻底实现。这是因为算法 1-1 中,对线性表 *A* 中元素的类型没有任何限制,而程序 1-1 却将其限制为整型数组。这样,如果要在一个浮点型数组中查找特定关键值就必须重写这个函数将其中的参数 a 和 x 的类型进行变换。这样对不同的类型都需要重写这个函数,将导致巨大的代码量和更多重复劳动。此外,在 C 语言中还要避免这些函数的同名冲突(相信读者初识 C 时都遭遇过对 abs 函数和 fabs 函数的困惑)。令代码具有通用性,使其尽可能地适用于各种可能的场合,是优秀程序员的追求。将

要处理的数据类型加以抽象,使其适用于不同类型的数据,是提高代码通用性的重要手段。各种语言抽象数据类型的所用的技术特点是有所不同的。就实现算法 1-1 为例,讨论 C 语言对数据类型抽象的方法。

C 语言数据抽象的利器——void 型指针

C 语言数据类型的抽象工具是 **void** 型指针。**void** 型指针可以指向存放任何类型数据的内存单元(组)。利用这一特性,用一个 **void** 型指针表示一个抽象元素类型的数组首元素地址,同时将特定关键值存放在一个变量中并将该变量的地址作为一个 **void** 型指针传递给函数。这样,似乎可将程序 1-1 中的 C 函数的原型声明改写成:

```
int linearSearch(void * a,int n,void * x);
```

然而,事情并没有那么简单:由于函数 linearSearch 调用时传递给它的实际参数 a 中元素的类型编译程序并不知道,所以如何用下标确定 a 中元素的地址还需要补充一点信息——数组中每个元素的存储宽度。这可以利用参数向函数 linearSearch 传递这一信息,于是函数 linearSearch 的原型声明写成:

```
int linearSearch(void * a,int size,int n,void * x);
```

其中,第二个参数 size 表示存储在数组 a 中的元素的存储宽度的。这样,就可通过地址 a+i * size 访问数组 a 中的第 i 个元素。然而,这样还不够,由于算法需要依次比较数组元素与特定值是否相等,而编译程序对两个 **void** 型指针 a+i * size 和 x 指向的内存中的数据无法执行比较运算,所以调用函数 linearSearch 时必须告诉它,如何比较数组中的元素与 x 指向的数据。这必须通过一个比较函数来完成,可以通过函数指针参数,把这一比较规则告诉函数 linearSearch。于是函数 linearSearch 写成:

```
1 int linearSearch(void * a,int size,int n,void * x,int( * comp)(void * ,void * )){
2     int i=0;
3     while(i<n){
4         if(comp((char * )a+i * size,x)==0)    /* 等价于 a[i]==x) */
5             return i;                         /* 找到关键值为 x 的元素,其下标为 i */
6         i++;
7     }
8     return -1;                                /* 未找到关键值为 x 的元素 */
9 }
```

程序 1-2　实现算法 1-1 在数组中查找元素的通用 C 代码

为便于代码重用,将程序 1-1 的源代码文件 search.c 连同声明函数 linearSearch 的头文件 search.h 存放在 utility 文件夹中。

该函数的实现要点中的第 1 点参数与返回值已经在前面讨论过了,第 2 点数据设置与程序 1-1 中的一样。此处,重点讨论第 3 点:关键代码。比较程序 1-1 与程序 1-2 的函数体可以发现,两者只有第 4 行中 **if** 语句中的条件表达式有所不同。在程序 1-1 中,该分支语句的条件表达式为 a[i]==x,而在程序 1-2 中表达式为 comp(a+i * size,x)==0,比较两者可见 comp(a+i * size,x)==0 替代了 a[i]==x。comp 是由参数传递进来的函数指针,它是指向一个具有两个 **void** 型指针参数且返回值为整数的函数的指针。我们约定:若第一个

指针指向的数据大于第二个指针指向的数据，该函数返回一个大于 0 的整数；若第一个指针指向的数据小于第二个指针指向的数据，该函数返回一个小于 0 的整数；而当两个指针指向的数据相等时，该函数返回 0。按此约定，条件 a[i]==x 就等价于 comp(a+i*size,x)==0。

可以对需要比较的数据类型事先定义好各自的比较函数，例如，下列代码就定义了比较两个由指针指向的整型、字符型、单精度浮点型和双精度浮点型数据的函数。

```
int intGreater(int *x,int *y){            /* 比较两个 int 指针指向的整型数据 */
    return (*x)-(*y);
}
int charGreater(char *x,char *y){         /* 比较两个 char 指针指向的字符型数据 */
    return (*x)-(*y);
  }
int floatGreater(float *x,float *y){      /* 比较两个 float 指针指向的单精度浮点数据 */
    if((*x-*y)>0.0)
        return 1;
    if((*x-*y)<0.0)
        return -1;
    return 0;
}
int doubleGreater(double *x,double *y){ /* 比较两个 double 指针指向的双精度浮点数据 */
    if((*x-*y)>0.0)
        return 1;
    if((*x-*y)<0.0)
        return -1;
    return 0;
}
```

程序 1-3　用两个指针指向的数据的比较函数

利用这些比较函数，就可以调用 linearSearch 在各种类型的数据中查找特定值了。例如，设 a 为 **double** 型数组，有 n 个元素，变量 x 中存有 **double** 型数据，调用程序 1-2 中的 linearSearch 函数在 a 中查找 x，调用形式为

```
linearSearch(a,sizeof(double),n,&x,doubleGreater);
```

若数组 a 的元素类型和变量 x 的类型为 **char**，则调用形式应为

```
linearSearch(a,sizeof(char),n,&x,charGreater);
```

为便于代码重用，将上述程序 1-3 存储为 utility 文件夹中的源文件 compare.c，并将这些函数的声明存储到同一文件夹内的头文件 compare.h 中。

1.4　数据的输入输出

1.4.1　应用问题

本节用上述实现的查找函数解决如下问题。

搜 索 引 擎

问题描述

Internet 以海量信息供用户查询而得到广泛应用。各大门户网站取胜的法宝之一就是强大的搜索引擎。搜索引擎用用户输入的关键值在大量的数据中查找相关信息。从某种意义上说,前面实现的 LINEAR-SEARCH 算法也是一个“搜索引擎”,它可以在一个存储了很多信息的线性表中查找特定的关键值。对待每一个搜索任务——一个关键值 x 和一组数据$A[p..q]$,若 A 中有与 x 相关的信息,则返回信息在表中的位置,否则给出“未找到”信息。

输入

输入文件 inputdata.txt 中记录了若干个搜索任务,每个任务表示成两行,第一行有两个数据项:第一个是一个字符:‘i’、‘c’、‘f’、‘s’分别表示关键值的类型为整数、字符、浮点数和字符串;第二个是对应类型的关键值。第二行包括若干个(最多不超过 80 个)对应类型的数据,数据项之间用一个空格隔开。

输出

输入文件中的每一个搜索任务,对应输出文件 outputdata.txt 中的一行信息:若数据表中有待查的信息,则输出关键值在数据表中的位置,否则输出信息“not found!”。

输入样例

```
i 35
2 9 24 35 78 56 90
f 12.0
20 32.1 27 0.35 12 24
c h
a b c d e f g h i j k l m n o p q r s t u v w x y z
s Beijing
Beijing Aomen Xianggang Chongqing Tianjin Shanghai
i 10
0 1 2 3 4 5 6 7 8 9
```

输出样例

```
4
5
8
1
not found!
```

问题分析

由于输入文件中对应每个搜索任务的数据项多少不一,解决这个问题的思路是按行读取输入文件的数据,每个搜索任务读取两次:读取的第一行包含数据类型信息 $type$ 和查找关键值 key,根据数据类型 $type$ 设法将包含于第二行中的若干该类型的数据存放到一个数组 $A[1..n]$中,然后调用过程 LINEAR-SEARCH 在 A 中查找key,若返回值 $i>0$,将 i 作为一

行写进输出文件，否则在输出文件中写入一行“not found!”。往返重复，直至输入文件结束。算法用伪代码表示如下。

```
SEARCH-ENGINE(f1,f2)          ▷f1 表示输入文件，f2 表示输出文件
打开输入文件 f1
打开输出文件 f2
while not end of  f1
  do first-str←从 f1 读入一行文本
     type←从 first-str 提取第一个字符
     key←按 type 指定的类型从 first-str 提取关键值
     second-str←从 f1 读入一行文本
     i←SEARCH(key,second-str)
     if i>0
       then 将 i 作为一行写入 f2
       else 将'hot found!'作为一行写入 f2
关闭文件 f1
关闭文件 f2
```

算法 1-3　解决搜索引擎问题的算法 SEARCH-ENGINE 过程

```
SEARCH(key,second-str)
n←0
while second-str 中还有数据
  do x←从 second-str 中提取一项数据
     n←n+1
     A[n]←x
return LINEAR-SEARCH(A,1,n,key)
```

算法 1-4　算法过程 SEARCH-ENGINE 中调用的 SEARCH 过程

解决本问题的算法很简单，但是实现算法会遇到如下的问题：输入数据中每一个搜索任务类型并不统一，所含的数据个数也不统一。换句话说，读取文件的操作需有针对不同数据类型和数据量的灵活性。

1.4.2　标准输入输出

C 语言将所有与程序外的数据交流统一视为“数据流”，程序外承载数据的集合统称为“文件”。计算机系统有两个特殊的文件：键盘——标准输入文件、屏幕——标准输出文件。本节以解决“搜索引擎”问题为例，说明数据输入输出技术，包括标准输入输出文件和磁盘文件的输入输出操作。

C 语言的头文件 stdio. h 中声明了若干个用于读写标准输入输出文件的库函数，如表 1-1 所示。

表 1-1　用于读写标准输入输出文件的库函数

函　　数	解　　释
scanf(格式串,变量地址表)	格式串中包含若干个格式符——以%引导的表示数据输入格式的符号,格式符与变量地址表中的项一一对应,在键盘上接受对各变量输入指定格式(类型)的数据
printf(格式串,表达式表)	格式串包含若干个格式符,格式符与表达式表中的项一一对应,将各表达式按指定格式(类型)依次向屏幕输出
getchar()	在键盘上接受一个字符的输入并作为返回值返回
puchar(字符表达式)	将作为参数的字符表达式的值输出到屏幕上
gets()	在键盘上接受一行字符的输入并将其作为一个字符串返回
puts(串表达式)	将作为参数的字符串表达式作为一行输出到屏幕

利用表 1-1 中罗列的各库函数还不足以解决本问题,因为当用 gets 从键盘上接受一行文本后,对如何从中正确提取信息 C 语言提供了读取包含于该文本行中的数据的库函数,如表 1-2 所示。

表 1-2　C 语言用于串输入输出操作的函数

函　数　名	解　　释
sscanf(文本串,格式串,变量地址表)	从文本串中接受对各变量输入指定格式(类型)的数据
sprintf(缓冲区串,格式串,表达式表)	将各表达式按指定格式(类型)依次向缓冲区串输出

然而,用 sscanf 从一个文本串中连续读取多个数据必须一次性传递指定个数的变量地址而不能通过每次读取一个多次循环重复来实现,因为每次重复,sscanf 都将从文本串的首地址开始读取数据。这给编程带来了困难:编程时可能并不知道需要在文本行中读取多少个数据。于是,需对 sscanf 进行一次拓展,定义如下的串输入流类型,并且声明对串输入对象的常用操作函数。

```
typedef struct{
    char * begin;                                      /* 输入流首地址 */
    char * current;                                    /* 输入流当前读取位置 */
}StrInputStream;                                       /* 串输入流 */
void initStrInputStream(StrInputStream * ,char * );    /* 初始化输入流 */
void sisRewind(StrInputStream * );                     /* 输入流还原初始设置 */
int sisEof(StrInputStream * );                         /* 检测输入流是否结束 */
int readInt(StrInputStream * ,int * );                 /* 从输入流中读取整数 */
int readDouble(StrInputStream * ,double * );           /* 从输入流中读取双精度浮点数 */
int readChar(StrInputStream * ,char * );               /* 从输入流中读取字符 */
int readString(StrInputStream * ,char * );             /* 从输入流中读取字符串 */
```

程序 1-4　串输入流类型定义以及串输入流操作函数的声明

对程序 1-4 的说明如下。

(1) 第 1～4 行定义的是串输入流 StrInputStream。它含有两个字符型指针属性：begin 和 current，分别指向串流首和当前读取数据的位置。

(2) 第 5～7 行分别声明了用来初始化输入流、还原输入流初始状态、判别输入流是否为空的常规维护函数 initStrInputStream、sisRewind、sisEof。

(3) 第 8～11 行声明了用来从输入流中读取整型数据函数 readInt、读取浮点型数据函数 readDouble、读取字符型数据函数 readChar 和读取字符串数据函数 readString。

为便于代码重用，将程序第 1～4 行中的代码保存在 utility 文件夹中的头文件 strstream.h 中。

下面定义程序 1-4 中声明的各函数。

1. 串输入流的常规维护

用来创建串输入流、判断输入流是否为空及恢复输入流初始状态的常规维护操作函数定义如下。

```
1 void initStrInputStream(StrInputStream * ssin,char * s){   /* 初始化输入流 */
2     while( * s==' '|| * s=='\t')                          /* 掠过空格 */
3         s++;
4     ssin->begin=ssin->current=s;
5 }
6 int sisEof(StrInputStream * ssin){
7     return (strlen(ssin->current)==0) || ( * (ssin->current)=='\r' ||
          ( * (ssin-> current)=='\n'));
8 }
9 void sisRewind(StrInputStream * ssin){              /* 还原输入流的初始状态 */
10    ssin->current=ssin->begin;
11 }
```

程序 1-5　定义串输入流常规维护操作函数的 C 代码

对程序 1-5 的说明如下。

(1) 第 1～5 行定义的函数 initStrInputStream 用串参数 s 创建一个 StrInputStream 对象 ssin。假定输入流中的数据项之间用空格作为分隔符。初始时，作为流载体的串可能存在前导空格，这会影响数据的正确读出，于是第 2～3 行的 **while** 循环负责掠过前导空格，第 4 行将串输入流的起始指针 begin 和当前读取指针 current 初始化为掠过了前导空格后的串 s。

(2) 第 6～8 行定义的函数 sisEof，通过检测串输入流参数 ssin 中 current 指向的串是否为空（串长度为 0）或 current 指向的字符为回车或换行符来判定串输入流是否为空。若串输入流 ssin 为空返回 1，否则返回 0。

(3) 第 9～11 行定义的函数 sisRewind，通过将串输入流参数 ssin 中的 current 指针赋值为 begin，来恢复输入流的初始状态。

2. 从串输入流中读取数据

从串输入流中可以读出各种基本数据类型的数据。操作步骤大同小异，下面仅就读取整型数据的函数展开讨论，读取其他类型数据的函数读者可打开本书提供的源代码中相应

文件研读。

```
1 int readInt(StrInputStream * ssin,int * x){
2     char s[80];
3     int n;
4     while( * (ssin->current)==' '|| * (ssin->current)=='\t')      /* 掠过空格 */
5         (ssin->current)++;
6     sscanf(ssin->current,"%s",s);        /* 从当前位置读取数据项 */
7     n=strlen(s);
8     if(!n)                               /* 未读到数据 */
9         return 0;
10    ssin->current+=n;                    /* 读到数据,调整读取位置 */
11    * x=atoi(s);                         /* 将数据转换成整数 */
12    return 1;
13 }
```

程序 1-6　从串输入流中读取整型数据的 C 函数定义

对程序 1-6 的说明如下。

(1) 函数 readInt 有两个参数。参数 ssin 表示一个指向串输入流的指针,参数 x 是指向接受数据的整型变量的指针。读取操作成功函数返回 1,否则返回 0。

(2) 函数内声明的局部变量 s 用来接受从输入流中以串格式读取的数据项。n 用来表示数据项的宽度。

(3) 第 4～5 行的 **while** 循环负责掠过前导空格,保证数据读取的正确性。第 6 行调用库函数 sscanf,从 current 指向的位置开始,以串格式(%s)将数据项读取到 s 中。第 7 行计算数据项的宽度 n,若 n 为 0 则说明读取数据项失败,第 9 行返回 0。读取成功时,n 为正整数,将 current 向后移动 n 个字节作为下一次从输入流中读取数据的位置(可能含有前导空格)。由此可见,将数据项先以串格式读取,可以正确计算其宽度 n,以此确定下一个读取位置 current。第 11 行调用库函数 atoi 将存储在 s 中的数据转换成整型数赋值给 x 指向的动态变量。读取正确,第 12 行返回 1。

利用程序 1-2、程序 1-4、程序 1-5 和程序 1-6,可以编出下列程序来解决搜索引擎问题。

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include "../../utility/strstream.h"
#include "../../utility/compare.h"
#include "../../utility/search.h"
int main(int argc,char** argv) {
    char first[80];
    gets(first);                          /* 读出第 1 个搜索任务的第 1 行 */
    while(strlen(first)){                 /* 只要有任务 */
        int i;
        StrInputStream sin;
```

```
    char type,second[250];
    initStrInputStream(&sin,first);          /* 用搜索任务的第 1 行创建一个串流 */
    readChar(&sin,&type);                    /* 读取串流中的类型信息 */
    switch(type){
        case 'i':{                           /* 数据类型为整数 */
          int key,a[80],n=0;
          readInt(&sin,&key);                /* 读取关键值 */
          gets(second);
          initStrInputStream(&sin,second);   /* 为搜索数据创建串输入流 */
          while(!sisEof(&sin))
              readInt(&sin,&a[n++]);         /* 从流中读取整型数据 */
          i=linearSearch(a,sizeof(int),n,&key,intGreater);
          break;
        }
        case 'f':{                           /* 数据类型为浮点数 */
            double key,a[80];
            int n=0;
            readDouble(&sin,&key);
            gets(second);
            initStrInputStream(&sin,second);
            while(!sisEof(&sin))
                readDouble(&sin,&a[n++]);
            i=linearSearch(a,sizeof(double),n,&key,doubleGreater);
            break;
        }
        case 'c':{                           /* 数据类型为字符 */
            char key,a[80];
            int n=0;
            readChar(&sin,&key);
            gets(second);
            initStrInputStream(&sin,second);
            while(!sisEof(&sin))
                readChar(&sin,&a[n++]);
            i=linearSearch(a,sizeof(char),n,&key,charGreater);
            break;
        }
        case 's':{                           /* 数据类型为字符串 */
            char *key,*a[80];
            int n=0,j;
            key=(char *)malloc(80*sizeof(char));     /* 为 key 分配空间 */
            for(j=0;j<80;j++)                /* 为数组 a 中元素分配空间 */
                a[j]=(char *)malloc(80*sizeof(char));
            readString(&sin,key);            /* 读取关键值 */
            gets(second);
            initStrInputStream(&sin,second);
```

```
58                  while(!sisEof(&sin))
59                      readString(&sin,a[n++]);
60                  i=linearSearch(a,sizeof(char *),n,key,strcmp);
61                  free(key);                  /* 释放 key 指引的空间 */
62                  for(j=0;j<80;j++)           /* 释放数组 a 中每个元素指引的空间 */
63                      free(a[j]);
64              }
65          }
66          if(i>-1)                            /* 搜索到 */
67              printf("%d\n",i);
68          else
69              printf("not found!\n");
70          gets(first);                        /* 读取下一个搜索任务的第 1 行 */
71      }
72      return (EXIT_SUCCESS);
73 }
```

程序 1-7 实现解决搜索引擎问题的算法 1-3 的 C 源代码文件

对程序 1-7 的说明如下。

(1) 第 7～73 行的 mian 函数实现算法 1-3。其中,第 10～71 行的 **while** 循环实现算法 1-3 中第 3～12 行的 **while** 循环。其中,第 8～9 行用读取的表示搜索任务的第一行文本 first 创建一个串输入流 sin,以供后面的代码从中读取数据类型 type 和关键值 key。

(2) 第 16～65 行的 **switch** 语句实现算法 1-3 中第 6～9 行的根据不同类型在数据集合中查找关键值的操作。其中,第 17～26 行、第 27～37 行、第 38～48 行和第 49～64 行分别针对整型、浮点型、字符型和字符串型在表示数据集合的数组中 a 中查找关键值 key。

(3) 以 **switch** 语句中的第 17～26 行对应于 **case** 常量 i 的分语句块为例,说明查找过程的执行。第 19 行在 first 中读取整型关键值 key,第 20 行从键盘读取表示搜索任务的第二行文本 second,第 21 行用 second 重新初始化串输入流 sin。第 22 行和第 23 行的 **while** 循环读取其中的数据到数组 a 中。第 24 行调用程序 1-2 中的函数 linearSearch 实施在整型数组 a 中查找关键值的操作返回值赋予 i。

其余的各 **case** 常量对应的分语句块的操作细节可参照上述分析理解。第 66～69 行根据 **switch** 语句计算所得的 i,向屏幕输出搜索结果信息。

(4) 由于输入输出文件是标准控制台文件(键盘和屏幕),所以算法 1-3 中第 1 行和第 2 行的文件打开操作和第 13 和第 14 行的文件关闭操作都省略了。

1.4.3 文件输入输出

在 C 语言中定义了文件指针类型 FILE,程序中所使用的每一个的文件都必须对应一个 FILE 类型的指针变量。表 1-3 列出了 C 语言常用的文件操作函数。

表 1-3　C 语言常用的文件操作函数

函　　数	解　　释
fopen(文件名串,打开模式串)	按读写模式打开指定文件,返回文件指针
fclose(文件指针)	关闭文件指针指向的文件
feof(文件指针)	检测文件读写指针是否指向文件尾
fgetc(文件指针)	从指定文件的当前读取位置读取一个字符并将其返回
fputc(字符表达式,文件指针)	将字符表达式的值写到指定文件当前读写位置处
fgets(字符数组,行长度,文件指针)	从指定文件的当前读写位置开始读取一个不超过指定长度的文本行到指定字符数组中
fputs(文本行,文件指针)	将指定文本行写到指定文件的当前读写位置处
fscanf(文件指针,格式串,变量地址表)	从指定文件中按指定格式读取数据到指定变量地址
fprintf(文件指针,格式串,表达式表)	将指定表达式按指定格式写到指定文件中

利用文件操作函数,把解决搜索引擎问题的 C 程序 1-7 改写成文件操作版本。

```
#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<string.h>
#include "../../utility/strstream.h"
#include "../../utility/compare.h"
#include "../../utility/search.h"
int main(int argc,char** argv) {
    char first[80];
    FILE *f1=fopen("chap01/SearchEnging/inputdata.txt","r"),
         *f2=fopen("chap01/SearchEnging/outputdata.txt","w");
    assert(f1&&f2);
    fgets(first,80,f1);                     /*读出第 1 个搜索任务的第 1 行*/
    while(!feof(f1)){                       /*只要有任务*/
        int i;
        StrInputStream sin;
        char type,second[250];
        initStrInputStream(&sin,first);     /*用搜索任务的第 1 行创建一个串流*/
        readChar(&sin,&type);               /*读取串流中的类型信息*/
        switch(type){
            case 'i':{                      /*数据类型为整数*/
                int key,a[80],n=0;
                readInt(&sin,&key);         /*读取关键值*/
                fgets(second,80,f1);
                initStrInputStream(&sin,second);  /*为搜索数据创建串输入流*/
                while(!sisEof(&sin))
                    readInt(&sin,&a[n++]);  /*从流中读取整型数据*/
                i=linearSearch(a,sizeof(int),n,&key,intGreater);
                break;
```

```
30              }
31              case 'f':{                    /* 数据类型为浮点数 */
32              ⋮
33              }
34              case 'c':{                    /* 数据类型为字符 */
35              ⋮
36              }
37              case 's':{                    /* 数据类型为字符串 */
38          ⋮
39              }
40          }
41          if(i>-1)                          /* 搜索到 */
42              fprintf(f2,"%d\n",i);
43          else
44              fprintf(f2,"not found!\r\n");
45          fgets(first,80,f1);               /* 读取下一个搜索任务的第 1 行 */
46      }
47      fclose(f1); fclose(f2);
48      return (EXIT_SUCCESS);
49  }
```

程序 1-8 解决搜索引擎问题的 C 程序 1-7 的文件操作版本

对程序 1-8 的说明如下。

(1) 第 10 行和第 11 行声明了两个文件指针型变量 f1 和 f2,并分别被赋予打开的输入文件 inputdata.txt 与输出文件 outputdata.txt 的指针。第 12 行调用标准库函数 assert 检测这两个文件是否成功打开。该函数的原型为

```
assert(condition)
```

当 condition 的值为 0 时,系统将显示 condition 不成立的信息,并退出程序的执行,这是 C 语言重要的异常处理方式。该函数的原型声明于头文件＜assert.h＞中。

(2) 第 13 行、第 24 行和第 45 行分别用 fgets 的调用替换程序 1-7 中相应位置上的 gets 的调用,在文件 f1 中读取文本行,而不是在键盘上读取。第 42 行和第 44 行分别用 fprintf 的调用替换程序 1-7 中相应位置上的 printf 的调用,将数据写到文件 f2 中,而不是写到屏幕上。

(3) 第 47 行分别关闭文件 f1 和 f2。

(4) 为节约篇幅,第 31 行、第 34 行和第 37 行的省略号表示程序 1～7 中相应位置的语句块,注意其中要将对 gets 的调用替换为 fgets 的调用。

程序 1-8 的源文件存储在文件夹 chap01/SearchEnging 中,读者可打开研读。

1.5 计数问题

数学是科学的皇后,而数学起源于计数。人们的生活离不开计数,相信很多读者的童年都是在念着"一、二、三、四、五,上山打老虎……"的童谣中度过的。简单的计数问题只需要

用眼：三个人，五本书……一眼就可看出。

1.5.1 简单模拟

问题涉及一个计数过程时，可以模拟这个过程实现计数。例如，下列由著名的 Joseph 问题演变而来的问题就可以通过这个方法加以解决。

Joseph

The Joseph's problem is notoriously known. For those who are not familiar with the original problem: from among n people, numbered $1,2,\cdots,n$, standing in circle every mth is going to be executed and only the life of the last remaining person will be saved. Joseph was smart enough to choose the position of the last remaining person, thus saving his life to give us the message about the incident. For example when $n=6$ and $m=5$ then the people will be executed in the order 5,4,6,2,3 and 1 will be saved.

Suppose that there are k good guys and k bad guys. In the circle the first k are good guys and the last k bad guys. You have to determine such minimal m that all the bad guys will be executed before the first good guy.

Input

The input consists of separate lines containing k. The last line in the input contains 0. You can suppose that $0<k<14$.

Output

The output will consist of separate lines containing m corresponding to k in the input.

Sample Input

```
3
4
0
```

Sample Output

```
5
30
```

1. 问题的描述与分析

Joseph 问题的原始版本大意是：对给定的两个正整数 n 和 m，编号为 $1\sim n$ 的 n 个人围坐一圈，从 1 号起连续报数，报到 m 者出局。剩下的人从当前位置开始从 1 起报数，报到 m 者出局……循环往复，直至剩下最后一个人。问出局者顺序如何？剩下者的原始编号是几？例如，$n=6$，$m=5$ 时，该过程出局者顺序为 5，4，6，2，3，剩下者的原始编号为 1。

本问题是，给定正整数 $k(<14)$，$n=2k$ 个人分成前后两组，编号分别为 $1\sim k$ 和 $k+1\sim n$。要求计算最小的 m，使得按 Joseph 游戏规则，最先出局的 k 个人恰为后面一组的。

问题可形式化地描述如下。

输入：正整数 k。

输出：最小的正整数 m，使得编号为 $1\sim n=2k$ 围坐一圈的 n 个人，循环地依次从 1 报到数 m，报到 m 者出局，前 k 个出局者恰为原始编号为 $k+1\sim n$ 的那些人。

2. 算法的伪代码描述

通过模拟这个游戏过程来对给定的 k 计算最小的 m。为便于数学计算，把 n 个人的编号设为 $0\sim k-1$，$k\sim n-1$。这样问题转换为如下。

输入：正整数 k。

输出：最小的正整数 m，使得编号为 $0\sim n-1$，围坐一圈的 $n=2k$ 个人，循环地依次报数 1 到 m，报到 m 者出局，前 k 个出局者恰为原始编号为 $k\sim n-1$ 的那些人。

首先，注意到 m 应为形如 $ak+b$ 形式的整数。其中，a 为奇数，$1\leqslant b\leqslant k$。这是因为，第 1 轮报数到 m 者的编号 t 应满足 $k\leqslant t<n$。对所有可能的 a 和 b 从小到大模拟 Joseph 游戏过程，对 $m=ak+b$，从第 1 轮的报数起始者编号 $t=0$ 开始，计算 $t+m-1$ 除以 n 的余数[①]，它表示在围坐一圈的 n 个人中，从 t 开始依次从 1 数到 m，最后一个人当时的编号。如果这个值小于 k 则表示目前的 m 不对，尝试下一个更大一点的 m，否则，n 减小 1 表示将编号大于 $k-1$ 的某个人出局，进入下一轮报数。若通过 k 轮报数都使得出局的人编号大于 $k-1$ 则得到正确的 m。将上述思想写成如下的伪代码过程。

```
JOSEPH(k)
a←1
while true
  do for b←1 to k
    do m←ak+b,t←0,n←2k
      for i←1 to k                    ▷尝试 k 轮报数
        do t←(t+m-1) MOD n            ▷计算本轮报数到 m-1 的人的位置
          if t<k then 尝试下一个 b 或 a  ▷m 不合法
          n←n-1
      if i>k then return m            ▷本 m 通过所有 k 轮报数
    a←a+2
```

算法 1-5 解决输入为 k 的 Joseph 问题的算法过程

程序实现

在 C 语言中实现算法 1-5 进而解决 Joseph 问题的代码如下。

```
int joseph(int k){
  int m,t,a=1,b,i,n;
  while(1){
    for(b=1;b<=k;b++){
      m=a*k+b;t=0,n=2*k;
      for(i=1;i<=k;i++){
```

① 在数学中，计算整数 x 除以 y 的余数称为模运算，记为 x MOD y。

```
7           t=(t+m-1)%n;
8           if(t<k)                   /* m 不合法,尝试下一个 b 或 a */
9               break;
10          n--;
11        }
12        if(i>k)                     /* 本 m 通过所有 k 轮报数 */
13          return m;
14      }
15      a+=2;
16  }
17 }
18 int main(){
19  int k;
20  FILE *f1=fopen("chap01/Joseph/inputdata.txt","r"),
21        *f2=fopen("chap01/Joseph/outputdata.txt","w");
22   assert(f1&&f2);
23   fscanf(f1,"%d",&k);
24   while(k){
25       fprintf(f2,"%d\n",joseph(k));
26       fscanf(f1,"%d",&k);
27   }
28   fclose(f1);fclose(f2);
29   return 0;
30 }
```

程序 1-9　解决 Joseph 问题的 C 程序

对程序 1-9 的说明如下。

(1) 第 1～17 行定义的函数 joseph 实现算法 1-5 的 JOSEPH 过程。代码结构与算法过程的伪代码十分接近,读者需要注意的是 C 语言中整数的模运算 a MOD b 是通过运算符%来完成的:a%b。

(2) 第 18～30 行的 main 函数调用函数 joseph 解决 Joseph 问题。第 20～22 行打开输入输出文件 f1、f2。

(3) 第 23～27 行依次处理输入文件中的每一个案例。其中,第 23、26 行从 f1 中读取案例数据,第 25 行调用函数 joseph 计算最小的报数值 m 并写入输出文件 f2。

程序 1-9 存储为文件夹 chap01/Joseph 中的源文件 joseph.c,读者可打开文件研读,并试运行。

1.5.2　加法原理和乘法原理

然而,复杂的计数问题就需要用心了。计数问题的基本方法是加法原理和乘法原理。

加法原理:做一件事,完成它可以有 n 类办法,在第一类办法中有 m_1 种不同的方法,在第二类办法中有 m_2 种不同的方法……在第 n 类办法中有 m_n 种不同的方法,那么完成这件

事共有 $N=m_1+m_2+m_3+\cdots+m_n$ 种不同方法。

乘法原理：做一件事，完成它需要分成 n 个步骤，做第一步有 m_1 种不同的方法，做第二步有 m_2 种不同的方法……做第 n 步有 m_n 种不同的方法，那么完成这件事共有 $N=m_1\times m_2\times m_3\times\cdots\times m_n$ 种不同的方法。

对于具体的计数问题，需要仔细考察完成计数有哪些方法，有多少步骤，合理运用加法原理和乘法原理得到问题的解。下面通过一个问题的分析解决来说明这两个原理的运用。

Escape

Description

Hamilton, the famous thief, plans a bank robbery in L. A. When searching for the escape route, he takes two main factors into consideration. First, he cannot pass through any intersection twice since the police will set up force at any intersection after he passes it for the first time. Second, as, in America, vehicles are driven on the right side, it is too risk for Hamilton to take a left turn at intersections when escaping. So he will always drive straight ahead or turn right when he comes to an intersection.

Hamilton is planning his escape route. He pays you, his partner, to calculate the number of different ways which leads him to his shelter. He gives you the map of the city, which looks like grids, with only streets leading in East-West direction and South-North direction. His starting position is (0,0) which is the South-West corner of the city and his shelter is located at (x, y) which stands for the intersection of the $(x+1)^{\text{th}}$ South-North direction street and the $(y+1)^{\text{th}}$ East-West direction street. Moreover, when he starts at (0,0), he is heading north, and of course, he can make a right turn there as well.

As the total number of different ways might be very large, you are asked to give the number's residue modulo 100000007.

Input

The first line of input contains $N(N\leqslant 100)$, the number of test cases. Each of the following lines describes one test case. Each line consists of four integers, X, Y, x, y $(0<X, Y\leqslant 2000, 0\leqslant x\leqslant X, 0\leqslant y\leqslant Y)$, which (X, Y) is the North-East corner of the city and (x, y) is the location of Hamilton's shelter. The famous thief, of course, cannot driver out of the city.

Output

For each test case, print one line with a single integer which is the corresponding answer.

Sample Input

```
3
3 4 0 0
3 4 1 0
3 4 1 1
```

Sample Output

1
13
16

1. 问题的描述与分析

城市有编号为 $0 \sim X$ 的 $X+1$ 条南北方向的街道，有编号为 $0 \sim Y$ 的 $Y+1$ 条东西方向的街道，这些街道共有 $(X+1) \times (Y+1)$ 个交叉处，如图 1-7 所示。把第 $i(0 \leqslant i \leqslant X)$ 条南北方向的街道与第 $j(0 \leqslant j \leqslant Y)$ 条东西方向的街道交叉处记为 (i, j)。左下角交叉点为 $(0, 0)$，而右上角交叉点为 (X, Y)。城市中的行车规则：只能直行或右转弯。给定 $(x, y)(0 \leqslant x \leqslant X, 0 \leqslant y \leqslant Y)$，计算从 $(0, 0)$ 出发到达 (x, y) 的行车路线数。

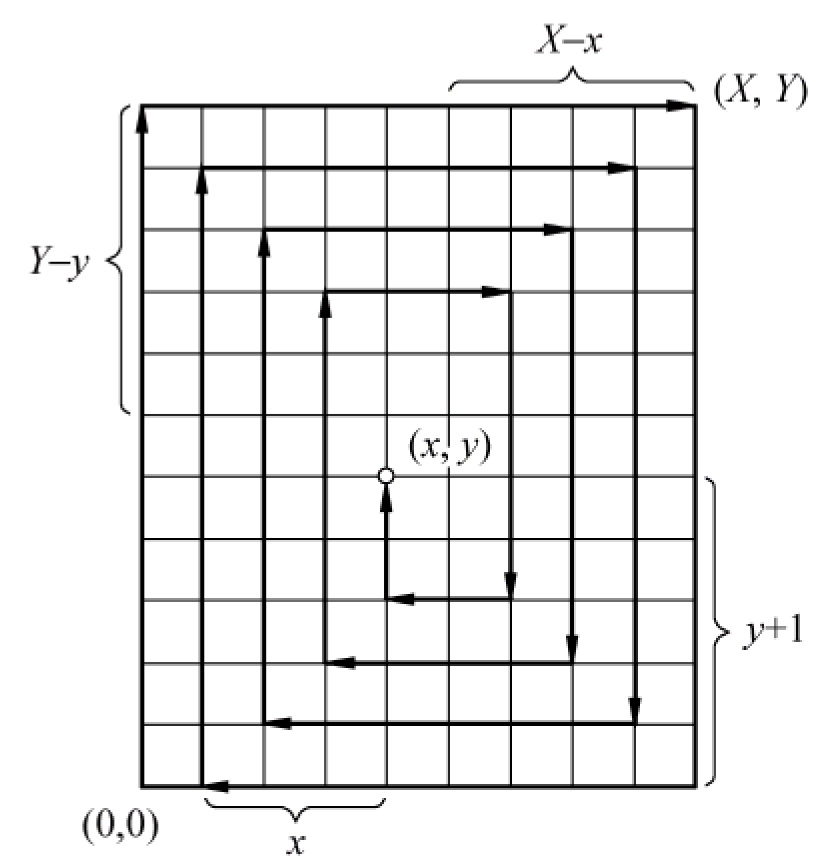


图 1-7　Hamilton 的逃跑路线

若 $x=0$，则只有 1 种行车路线——直行。对 $0<x \leqslant X, 0 \leqslant y \leqslant Y$ 的情形，设 $m=\min\{Y-y, X-x, y, x-1\}$。考虑从 $(0, 0)$ 处出发，按行车规则（只能直行或右转，且每个交叉口最多只能通过一次）到达 (x, y) 期间右转次数对路径分类计数。

设 $m=0$。

(1) 若 $Y-y=m$，转 1 次弯：从 $(0, 0)$ 直行到 $(0, y)$，右转，直行到 (x, y)。只有一条这样的路径。

(2) 转若 $X-x=m$，转 2 次弯：从 $(0, 0)$ 直行到 $(0, y+j)$，其中 $0<j \leqslant Y-y$。右转，直行到 $(x, y+j)$。右转，直行到 (x, y)。一共有 $Y-y$ 条这样的路径。

(3) 若 $y=m$，转 3 次弯：从 $(0, 0)$ 直行到 $(0, y+j)$，其中 $0<j \leqslant Y-y$。右转，直行到 $(x+i, y)$，其中 $0<i \leqslant X-x$。右转，直行到 (x, y)。有 $(Y-y)(X-x)$ 条这样的路径。

(4) 若 $x-1=m$，转 4 次弯：从 $(0, 0)$ 直行到 $(0, y+j)$，其中 $0<j \leqslant Y-y$。右转，直行到 $(x+i, y+j)$，其中 $0<i \leqslant X-x$。右转，直行到 $(x+i, y+j-j')$，其中 $0<j' \leqslant y$。右转，直行到 (x, y)。有 $(Y-y)(X-x)y$ 条这样的路径。

下设 $m=1$。

(5) 若 $Y-y=m$，转 5 次弯：从 $(0, 0)$ 直行到 $(0, y+j)$，其中 $0<j \leqslant Y-y$。右转，直行到 $(x+i, y+j)$，其中 $0<i \leqslant X-x$。右转，直行到 $(x+i, y+j-j')$，其中 $0<j' \leqslant y$。右转，直行到 $(x+i-i', y+j-j')$，其中 $0<i'<x$。右转，直行到 (x, y)。有 $(Y-y)(X-x)y(x-1)$ 条这样的路径。

(6) 若 $X-x=m$，转 6 次弯：$(Y-y)(X-x)y(x-1)$ 条转了 5 个弯的路径中，转第一个弯后的 $(Y-y)$ 条路径自上而下各有 $(Y-y-1), Y-y-2, \cdots, 2, 1, 0$ 种转第 6 个弯的可能，所以，根据加法原理有：

$$\begin{aligned}&((Y-y-1)+(Y-y-2)+\cdots+2+1)\ (X-x)y(x-1)\\=&((Y-y)(Y-y-1)/2\ )(X-x)y(x-1)\end{aligned}$$

条这样的路径。

(7) 若 $y=m$,转 7 次弯:与(6)的讨论相仿,有 $(Y-y)(Y-y-1)(X-x)(X-x-1)y(x-1)/2^2$ 条这样的路径。

(8) 若 $x-1=m$,转 8 次弯:有 $(Y-y)(Y-y-1)(X-x)(X-x-1)y(y-1)(x-1)/2^3$ 条这样的路径。

下设 $m=2$。

(9) 若 $Y-y=m$,转 9 次弯:有 $(Y-y)(Y-y-1)(X-x)(X-x-1)y(y-1)(x-1)(x-2)/2^4$ 条这样的路径。

⋮

一般地,对 $m>0$,

$Y-y=m$,有 $\frac{P^m_{Y-y}}{2^{m-1}}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$ 条转 $4m+1$ 次弯的路径。

$X-x=m$,有 $\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$ 条转 $4m+2$ 次弯的路径。

$y=m$,有 $\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^{m+1}_{X-x}}{2^m}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$ 条转 $4m+3$ 次弯的路径。

$x-1=m$,有 $\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^{m+1}_{X-x}}{2^m}\frac{P^{m+1}_y}{2^m}\frac{P^m_{x-1}}{2^{m-1}}$ 条转 $4(m+1)$ 次弯的路径。

其中 $P^m_n=\frac{n!}{(n-m)!}$。根据加法原理和乘法原理,将计算一个案例的过程表示成如下的伪代码。

ESCAPE(X, Y, x, y)
if $x=0$
  then return 1
$s\leftarrow 1+Y-y+(Y-y)(X-x)+(Y-y)(X-x)y+(Y-y)(X-x)y(x-1)$
$m\leftarrow \min\{X-x, Y-y, x-1, y\}$
if $m=0$
  then return s
for $j\leftarrow 1$ **to** $m-1$
  do $s\leftarrow s+\frac{P^j_{Y-y}}{2^{j-1}}\frac{P^j_{X-x}}{2^{j-1}}\frac{P^j_y}{2^{j-1}}\frac{P^j_{x-1}}{2^{j-1}}+\frac{P^{j+1}_{Y-y}}{2^j}\frac{P^j_{X-x}}{2^{j-1}}\frac{P^j_y}{2^{j-1}}\frac{P^j_{x-1}}{2^{j-1}}+\frac{P^{j+1}_{Y-y}}{2^j}\frac{P^{j+1}_{X-x}}{2^j}\frac{P^j_y}{2^{j-1}}\frac{P^j_{x-1}}{2^{j-1}}+\frac{P^{j+1}_{Y-y}}{2^j}\frac{P^{j+1}_{X-x}}{2^j}\frac{P^{j+1}_y}{2^j}\frac{P^j_{x-1}}{2^{j-1}}$
if $Y-y=m$
  then return $s+\frac{P^m_{Y-y}}{2^{m-1}}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$
if $X-x=m$
  then return $s+\frac{P^m_{Y-y}}{2^{m-1}}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}+\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$
if $y=m$
  then return $s+\frac{P^m_{Y-y}}{2^{m-1}}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}+\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^m_{X-x}}{2^{m-1}}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}+\frac{P^{m+1}_{Y-y}}{2^m}\frac{P^{m+1}_{X-x}}{2^m}\frac{P^m_y}{2^{m-1}}\frac{P^m_{x-1}}{2^{m-1}}$

return $s+\frac{P_{Y-y}^{m}}{2^{m-1}}\frac{P_{X-x}^{m}}{2^{m-1}}\frac{P_{y}^{m}}{2^{m-1}}\frac{P_{x-1}^{m}}{2^{m-1}}+\frac{P_{Y-y}^{m+1}}{2^{m}}\frac{P_{X-x}^{m}}{2^{m-1}}\frac{P_{y}^{m}}{2^{m-1}}\frac{P_{x-1}^{m}}{2^{m-1}}$

$$+\frac{P_{Y-y}^{m+1}}{2^{m}}\frac{P_{X-x}^{m+1}}{2^{m}}\frac{P_{y}^{m}}{2^{m-1}}\frac{P_{x-1}^{m}}{2^{m-1}}+\frac{P_{Y-y}^{m+1}}{2^{m}}\frac{P_{X-x}^{m+1}}{2^{m}}\frac{P_{y}^{m+1}}{2^{m}}\frac{P_{x-1}^{m}}{2^{m-1}}$$

2. 程序实现

利用上述对问题的认识与解决该问题的算法，很容易在 C 语言中编写出下列程序。

```
const int MOD=100000007;                    /* 这是一个素数 */
unsigned min(unsigned a, unsigned b, unsigned c, unsigned d){
  unsigned x=a<=b? a:b, y=c<=d? c:d;
  return x<=y? x:y;
}
unsigned escape(unsigned X,unsigned Y, unsigned x, unsigned y){
  unsigned long long f1=Y-y,f2=X-x,f3=y,f4=x-1,
      P1=f1, P2=f2, P3=f3, P4=f4,      /* P1、P2、P3、P4 分别表示计算中的 4 个排列数 */
      m=min(Y-y, X-x, y, x-1),j,s;
  if(x==0)
      return 1;
  if(m==0){
      if(Y-y==m)
          return 1;
      if(X-x==m)
          return 1+(unsigned)P1;
      if(y==m)
          return 1+(unsigned)P1+(unsigned)((P1 * P2)%MOD);
      return 1+(unsigned)P1+(unsigned)((P1 * P2)%MOD)+
                (unsigned)((((P1 * P2)%MOD) * P3)%MOD);
  }
  s=1+P1+(P1 * P2)%MOD+(((P1 * P2)%MOD) * P3)%MOD;
  for(j=1;j<m;j++){
      s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
      P1=(--f1 * P1/2)%MOD;/* 计算关于 Y-y 的排列数 */
      s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
      P2=(--f2 * P2/2)%MOD;/* 计算关于 X-x 的排列数 */
      s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
      P3=(--f3 * P3/2)%MOD;/* 计算关于 y 的排列数 */
      s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
      P4=(--f4 * P4)%MOD;/* 计算关于 x-1 的排列数 */
  }
  if(Y-y==m)
      return (unsigned)s+
             (unsigned)((((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD);
  if(X-x==m){
```

```
35         s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
36         P1=(--f1 * P1/2)%MOD;
37         return (unsigned)s+
                 (unsigned)((((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD);
38     }
39     if(y==m){
40         s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
41         P1=(--f1 * P1/2)%MOD;
42         s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
43         P2=(--f2 * P2/2)%MOD;
44         return (unsigned)s+
                 (unsigned)((((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD);
45     }
46     s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
47     P1=(--f1 * P1/2)%MOD;
48     s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
49     P2=(--f2 * P2/2)%MOD;
50     s=s+(((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD;
51     P3=(--f3 * P3/2)%MOD;
52     return(unsigned)s+(unsigned)((((P1 * P2)%MOD) * ((P3 * P4)%MOD))%MOD);
53 }
```

程序 1-10　解决 Escape 问题的 C 程序

对程序 1-10 的说明如下。

(1) 第 6～53 行定义的函数 escape 实现算法过程 ESCAPE(X, Y, x, y)对由参数表示的案例数据 X、Y、x、y 计算合法的逃逸线路数。其为了减少重复计算$\frac{P^k_{Y-y}}{2^{k-1}}$、$\frac{P^k_{X-x}}{2^{k-1}}$、$\frac{P^k_{y}}{2^{k-1}}$和$\frac{P^k_{x-1}}{2^{k-1}}$,在第 7 行设置这些式子中分子的当前因子 f1、f2、f3 和 f4,分别初始化为 Y-y、X-x、y 和 x-1。第 8 行设置 P1、P2、P3 和 P4 表示各分子,并分别初始化为 f1、f2、f3 和 f4。在后面的计算中每次累乘前将因子 fi(i=1,2,3,4)自减 1。

(2) 第 9 行调用第 2～5 行定义的函数 min,计算 Y-y、X-x、y 和 x-1 的最小值 m。第 10 行和第 11 行处理算法中 x=1 的情形,第 12～20 行处理 m=0 的情形。第 21～52 行处理一般的 m>0 的情形,实现算法过程中第 7～14 行的操作。其中,第 22～31 行的 **for** 循环实现算法过程中的第 7 行和第 8 行的操作。第 32～33 行、第 34～38 行、第 39～44 行及第 45～52 行分别对应算法过程中的第 9～10 行、第 11～12 行、第 13～14 行及第 15 行的操作。

(3) 由于两个非负整数的积可能超出 **unsigned** 类型的取值范围,故 P1、P2、P3、P4 及 f1、f2、f3、f4 均定义为 **usigned long long** 的 64 位无符号整型,并且在计算过程中,每次进行乘法运算后都对 MOD=100000007 进行求模运算,使得在后面的乘法计算中不会发生溢出错误。该常数定义于第 1 行。由于 100000007 是一个素数,所以所有小于该数的正整数对乘法构成一个群,即对于乘法运算是封闭的(参见本书 6.4.1 节)。

程序 1-10 的代码存储在文件夹 chap01/Escape 中的源文件 Escape.c 中，可打开文件研读并试运行。

1.5.3　计算四边形个数

计数问题有时是极富挑战性的，需要仔细考量问题中所涉及对象的数据模型，研究其中的各个细节，灵活运用所有的数学、科学知识和生活经验对其进行充分、深入的研究。例如，下面这个问题就需对其中的数据模型进行由表及里地分析，找出输入、输出数据之间的内在联系，进而得到解决问题的算法。解决复杂问题的过程往往是一个不断探索、思考、判断的过程，可以在这样的过程中获取成功的喜悦。

Counting Quadrangles

Description

Your task is to count how many quadrangles are there in this kind of picture（见图 1-8）(The following illustration is a picture whose size=7).

Input

There are several test cases in the input file. Each line contains a single number N, which is the size of the picture. $N=0$ indicates the end of input file.

Output

For each test case, output the number of the quadrangles in the format as indicated in the sample output. It's guaranteed that the number is less than 2^{63}.

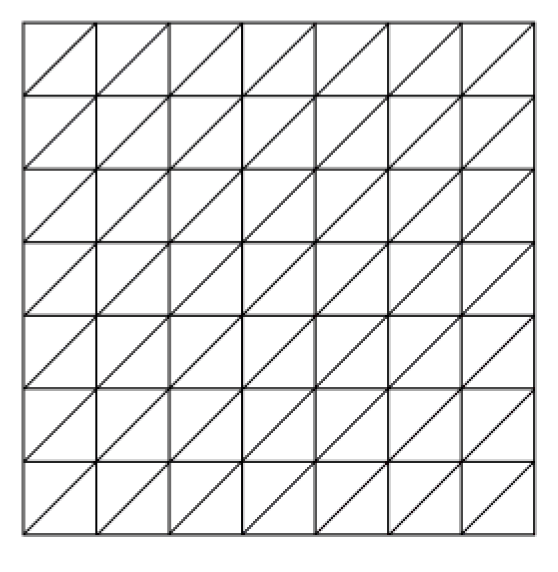


图 1-8　问题描述图

Sample Input

```
1
2
7
0
```

Sample Output

```
Case 1: 1
Case 2: 23
Case 3: 3108
```

1. 问题与算法描述

对输入的正整数 n，计算并输出由 n^2 个带有对角线的小方块构成的正方形图案中含有的凸四边形个数。如果仅用眼力解决这个问题，估计很多读者对图 1-9(a)中仅由 4 个小正方形构成的图案看不了多久就会眼花，所以此处得考虑用其他方法解决此问题。

首先，我们观察到，图案中的凸四边形有矩形（包括正方形）、竖直平行四边形（见

图 1-9(b))、水平平行四边形(见图 1-9(c))、竖直梯形(见图 1-9(d))、水平梯形(见图 1-9(e))、等腰梯形(如图 1-7(f))6 种。它们在图案中大小相异,相互并列、交错、包含。设最小正方形的边长为 1,通过考察图案中高度一定的各种四边图形的个数,利用加法原理计算四边形总数。

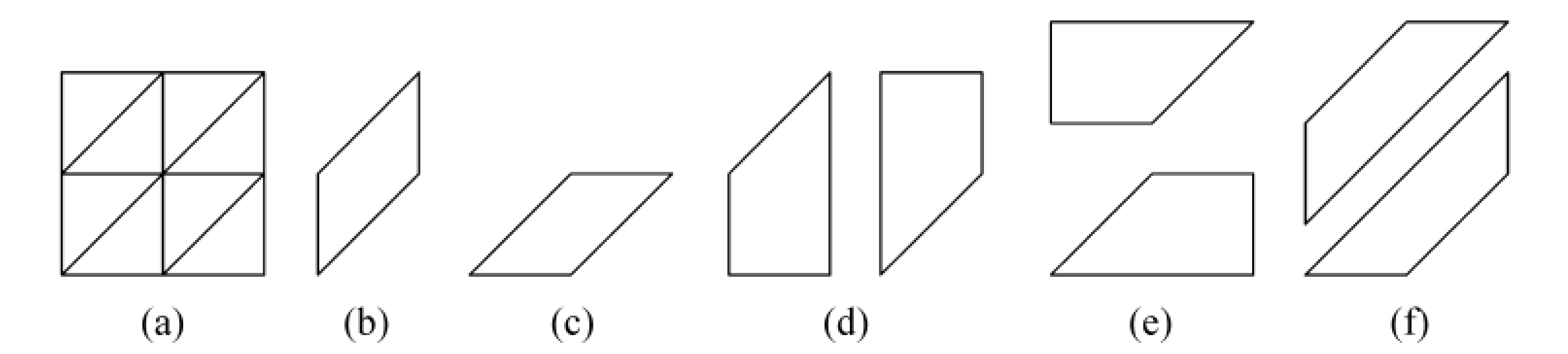


图 1-9　Counting Quadrangles 问题中的图案及其中所含的凸四边形

写成伪代码过程如下。

```
COUNTING-QUADRANGLES(n)
1 count←0
2 for k←1 to n
3    do count←count+图案中高度为 k 的四边形个数
4 return count
```

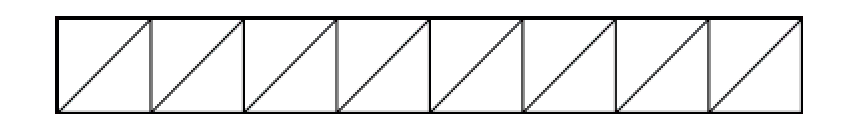
算法 1-6　解决 Counting Quadrangles 问题的过程

如果能在常数时间内计算出图案中高度为 $k(1\leqslant k\leqslant n)$ 的四边形数量,则算法 COUNTING-QUADRANGLES 的运行时间为 $\Theta(n)$。

2. 程序实现

要将算法 1-6 实现为程序,需确切地计算出图案中高度为 $k(1\leqslant k\leqslant n)$的四边形数量。为此,进行如下思考。

高度 k 为 1 时,含有 n 个并排的由 n 个小方块构成的图形,如图 1-10 所示。


图 1-10　由 n 个小方块构成的图形

在这个图形中,所含高度为 1 的四边形如表 1-4 所示。

表 1-4　n 个小方块组成的图形含有高度为 1 的各种四边形个数计算

图　　形	数　　量
矩形	$\sum_{i=1}^{n} i$ (宽度分别为 1,2,…,n)
竖直平行四边形	0
竖直梯形	0

续表

图　　形	数　　量
水平平行四边形	$\sum_{i=1}^{n-1} i$（宽度分别为 1,2,…,$n-1$）
水平梯形	$2\sum_{i=1}^{n-1} i$（宽度分别为 1,2,…,$n-1$。斜边分别位于左、右）
等腰梯形	0

根据加法原理，该图形含有 $\sum_{i=1}^{n} i + 3\sum_{i=1}^{n-1} i = n(n+1)/2 + 3(n-1)n/2 = 2n^2 - n$ 个不同的凸四边形。整个图案含有 n 个这样的图形，根据乘法原理，图案含有 $n(2n^2-n)=2n^3-n^2$ 个高度为 1 的凸四边形。

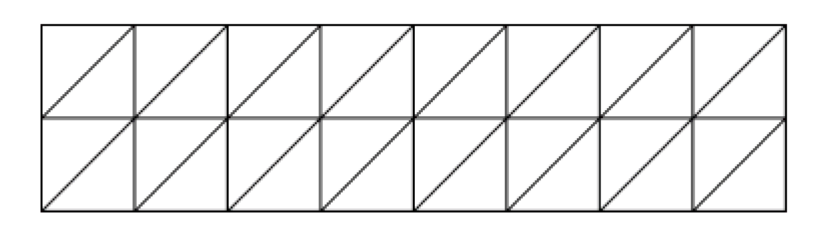


图 1-11　由 $2n$ 个小方块构成的图形

高度 k 为 2 的由 $2n$ 个小方块构成的图形，如图 1-11 所示。在这个图形中，所含的高度为 2 的四边形数量如表 1-5 所示。

即该图形含有 $\sum_{i=1}^{n} i + 3n + 3\sum_{i=1}^{n-2} i + 2(n-1)$ 个高度为 2 的四边形。这个图案有 $n-1$ 个这样的图形，所以有 $(n-1)\times(\sum_{i=1}^{n} i + 3n + 3\sum_{i=1}^{n} i + 2(n-1))$ 个高度为 2 的四边形。

表 1-5　$2n$ 个小方块组成的图形含有高度为 2 的各种四边形个数计算

图　　形	数　　量
矩形	$\sum_{i=1}^{n} i$（宽度分别为 1,2,…,n）
竖直平行四边形	n（仅有宽度为 1）
竖直梯形	$2n$（仅有宽度为 1，上下对称各一个）
水平平行四边形	$\sum_{i=1}^{n-2} i$（宽度分别为 1,2,…,$n-2$）
水平梯形	$2\sum_{i=1}^{n-2} i$（宽度分别为 1,2,…,$n-2$，左右对称各一）
等腰梯形	2(n−1)（每条长斜线对应两个对称的等腰梯形）

而高度为 3 的由 $2n$ 个小方块构成的图形，如图 1-12 所示。此图形中所含各种高度为 3 的各种四边形数量如表 1-6 所示。

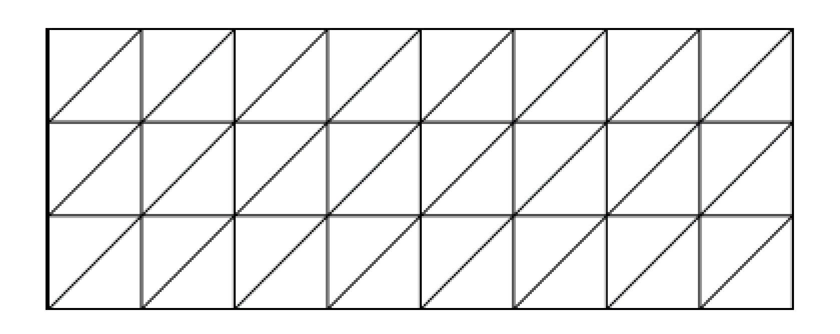


图 1-12　由 $3n$ 个小方块构成的图形

表 1-6　3*n* 个小方块组成的图形含有高度为 3 的各种四边形个数计算

图　　形	数　　量
矩形	$\sum_{i=1}^{n} i$(宽度分别为 $1,2,\cdots,n$)
竖直平行四边形	$n+(n-1)$ (宽度分别为 1,2)
竖直梯形	$2(n+(n-1))$ (宽度分别为 1,2,上下对称)
水平平行四边形	$\sum_{i=1}^{n-3} i$(宽度分别为 $1,2,\cdots,n-3$)
水平梯形	$2\sum_{i=1}^{n-3} i$(宽度分别为 $1,2,\cdots,n-3$,左右对称)
等腰梯形	$2(n-2)\times 2$(对称于每条 3×3 正方形对角线有 2×2 个等腰梯形)

根据加法原理,图形中含有 $\sum_{i=1}^{n} i+3(n+(n-1))+3\sum_{i=1}^{n-3} i+2(n-2)\times 2$ 个高度为 k 的四边形。图案中共有 $n-2$ 个这样的图形,根据乘法原理,图案中共有 $(n-2)(\sum_{i=1}^{n} i+3(n+(n-1))+3\sum_{i=1}^{n-3} i+2(n-2)\times 2)$ 个高度为 3 的四边形。

一般地,高度为 $k(1<k\leqslant n)$ 的由 $k\times n$ 个小方块构成的图形中,所含高度为 k 的各种四边形数量如表 1-7 所示。

表 1-7　*kn* 个小方块组成的图形含有高度为 *k* 的各种四边形个数计算

图　　形	数　　量
矩形	$\sum_{i=1}^{n} i$(宽度分别为 $1,2,\cdots,n$)
竖直平行四边形	$\sum_{i=n-(k-1)}^{n} i$(宽度分别为 $1,2,\cdots,k-1$)
竖直梯形	$2\sum_{i=n(k-1)}^{n} i$(宽度分别为 $1,2,\cdots,k-1$,上下对称)
水平平行四边形	$\sum_{i=1}^{n-k} i$(宽度分别为 $1,2,\cdots,n-k$)
水平梯形	$2\sum_{i=1}^{n-k} i$(宽度分别为 $1,2,\cdots,n-k$,左右对称)
等腰梯形	$2(n-k+1)\times(k-1)$(对称于每条 $k\times k$ 的斜线有 $2\times(k-1)$ 个等腰梯形)

根据加法原理,图形中含有 $\sum_{i=1}^{n} i+3\sum_{i=n-(k-1)}^{n} i+3\sum_{i=1}^{n-k} i+2(n-k+1)\times(k-1)$ 个高度为 k 的四边形。整个图案共有

$$
\begin{aligned}
&(n-k+1)\left(\sum_{i=1}^{n} i+3\sum_{i=n-(k-1)}^{n} i+3\sum_{i=1}^{n-k} i+2(n-k+1)\times(k-1)\right)\\
=&\ (n-k+1)(n(n+1)/2+3(n-k+1)(n-k)/2\\
&+3(k-1)(2n-k+2)/2+2(n-k+1)(k-1))
\end{aligned}
$$

个高度为 k 的四边形。其中，$1<k\leqslant n$。

将上述的计算结果用在算法 1-6 中，可得到下列解决 Counting Quadrangles 问题的 C 程序。

```
1 #include<stdio.h>
2 #include<assert.h>
3 int countingQuadrangles(int n){
4   int k,count=0;
5   count=n*(2*n*n-n);                      /*高为 1*/
6   for(k=2;k<=n;k++){                      /*高为 k*/
7     count+=(n-k+1)*(
8         n*(n+1)/2                         /*矩形*/
9         +3*(n-k+1)*(n-k)/2                /*水平平行四边形、竖直梯形*/
10        +3*(k-1)*(2*n-k+2)/2              /*竖直梯形、平行四边形*/
11        +2*(n-k+1)*(k-1)                  /*等腰梯形*/
12        );
13  }
14  return count;
15 }
16 int main(){
17  int n,i=1;
18  FILE *f1=fopen("chap01/Counting Quadrangles/inputdata.txt","r"),
19       *f2=fopen("chap01/Counting Quadrangles/outputdata.txt","w");
20  assert(f1&&f2);
21  fscanf(f1,"%d",&n);
22  while(n){
23    fprintf(f2,"Case %d: %d\n",i++,countingQuadrangles(n));
24    fscanf(f1,"%d",&n);
25  }
26  fclose(f1); fclose(f2);
27  return 0;
28 }
```

程序 1-11　解决 Counting Quadrangles 问题的 C 源代码文件 Counting Quadrangles.c

对程序 1-11 的说明如下。

（1）第 16～28 行定义的 main 函数中，第 21～25 行的 **while** 循环按题目中规定的输入文件的格式从文件指针 f1 指引的文件中逐一读出整型数据 n，第 23 行调用实现算法 1-6 的函数 countingQuadrangles(n)计算 n×n 个小方块构成的图案中包含的凸四边形的个数，并将计算结果作为一行写到文件指针 f2 指引的输出文件中。

（2）第 3～15 行定义的函数 countingQuadrangles 实现算法 1-6 的 COUNTING-QUADRANGLES。根据以上的分析，计算高度 k=1 的图形所含四边形个数的公式与 k>1 的情形不一致，所以第 5 行单独计算块时的情形，连同第 6～13 行的 **for** 循环，实现算法 1-6

中的第 2 行和第 3 行的 **for** 循环。将不同高度的四边形个数累加到变量 count 中。其中第 8～11 行分别计算高度为 k(1＜k≤n)的图形中所含的矩形、水平平行四边形和梯形、竖直平行四边形和梯形以及等腰梯形的个数。

源代码文件 CountingQuadrangles.c 存储在文件夹 chap01/Counting Quadrangles 中，读者可打开文件研读并试运行。

第 2 章　数据结构基础

算法就是输入数据和输出数据之间转换的计算步骤。如何组织输入数据和输出数据以及从前者到后者转化过程中的数据，将极大地影响算法的效率。在计算机科学中，数据在计算机中的组织、表示方法称为**数据结构**。无论是输入数据、输出数据还是算法内部需要维护的数据大多数情况下都可表示为一些集合。因此，集合既是数学的基础，也是计算机科学的基础。在输入到输出的转换过程中，算法所维护的集合通常需要随时进行增长、缩小及其他变化，把这样的集合称为是**动态**的。本章将介绍在计算机上表示及操作有限动态集合的一些基本技巧。各种算法要求对集合进行不同类型的操作。算法对集合的最基本的操作包括将元素插入到集合中，能从集合中删除元素，并可检测元素是否从属于集合等操作。一个支持这些操作的动态集合称为一个**字典**。

在计算机中要表示一个动态集合，集合中的每一个元素都要表示成只要具有指向它的指针就能检测及处理其各属性的对象。动态集合中的元素能分辨各自身份的属性称为**关键值**域。事实上集合的元素还可以含有**卫星数据**，这些数据加载于对象的其他域中但在集合的实现中并不使用。为阐述简单，本章讨论中将动态集合视为关键值的集合。在动态集合的实现中，元素对象还可能包含一些集合操作所要处理的域，这些域往往是指向其他元素对象的指针（或对象的引用）。

有些动态集合假定关键值取自于一个诸如实数集或由字母表顺序构成的单词集合这样的**全序集**（集合内任意两个元素均有且仅有等于、大于或小于 3 种关系之一）。一个全序集允许定义集合的最小元素，也允许集合中下一个元素大于给定的元素。

动态集合上的操作可以分成两类：直接返回集合的有关信息**查询类**操作和将改变集合的**变更类**操作。具体的应用通常都需实现下列操作中的部分或全部。

1. SEARCH(*S*, *k*)

这是一个查询类操作，给定集合 S 及一个关键值 k，返回一个指向 S 中某元素的指针 x 使得 $key[x]=k$，若 S 中没有这样的元素返回 NIL。

2. INSERT(*S*, *x*)

这是一个变更型操作，它在集合 S 中添加指向 x 的元素。通常假定元素 x 的所有属性域都已经初始化好了。

3. DELETE(*S*, *x*)

这是一个变更型操作，给定一个指向集合 S 中某个元素的指针 x，从 S 中移除 x（注意此操作使用的是指向元素 x 的指针，而不是关键值）。

4. MINIMUM(S)

这是一个对全序集 S 的查询类操作,返回指向 S 中关键值最小的元素的指针。

5. MAXIMUM(S)

这是一个对全序集 S 的查询类操作,返回指向 S 中关键值最大的元素的指针。

6. SUCCESSOR(S,x)

这是一个查询类操作,对给定的其关键值取自于一个全序集 S 的元素 x,返回一个指向 S 中比 x 大的下一个元素的指针,或若 x 是最大元素则返回 NIL。

7. PREDECESSOR(S,x)

这是一个查询类操作,对给定的其关键值取自于一个全序集 S 的元素 x,返回一个指向 S 中比 x 小的下一个元素的指针,或若 x 是最小元素则返回 NIL。

执行一个集合操作的时间通常表示为集合的大小 n 的函数 $T(n)$。

2.1 线性表

在对数据集合的各种操作中,几乎都要用到一个最基本的操作:依次考察部分或全部元素。这个操作通常称为对集合元素的**遍历**或**扫描**。为了有效地对集合元素进行扫描,很自然地将集合中的 n 个元素一字排列,如图 2-1 所示。

a_1, a_2, …, a_{i-1}, a_i, a_{i+1}, …, a_n

表首　　表尾

图 2-1　一个线性表

在图 2-1 的元素排列中,有且仅有一个元素 a_1 没有前驱,该元素称为**表头**;有且仅有一个元素 a_n 没有后继,称为**表尾**;此外的所有 $1<i<n$ 的元素 a_i 均有一个前驱 a_{i-1},且有一个后继 a_{i+1}。以这样的方式组织起来的数据结构称为**线性表**。

2.1.1 线性表的链表表示

在计算机中,用一个连续存储的数组来表示一个线性表是很自然的,因为数组的下标自然地表示出了线性表中元素的前后关系。连续存储指的是在内存中为每个元素分配同等大小的一块存储空间,相邻的存储空间存放相邻的两个元素。正由于数组与线性表下标的自然对应关系,在很多应用中都用数组来表示线性表。

然而,对线性表除了扫描以外还有很多其他操作,有些操作对数组来说就未必是很方便了。例如,如果要对数组中的元素进行增删操作,每一次这样的操作,都会引起前后元素的移动,这是很费时、费事的。为提高这类操作的效率,在计算机中,可以用非连续存储方式来表示线性表,也就是链表方法。

链表中为每一个元素单独分配一个称为**结点**的存储空间,元素间的前后顺序是由指向

每个结点的指针确定的。如图 2-2 所示，**双向链表** *L* 的每一个结点具有一个称为 *key* 的数据域，两个分别称为 *next* 和 *prev* 的指针域的对象。给定一个表中的元素 *x*，*next*[*x*]指向其在表中的后继，而 *prev*[*x*]指向其前驱。若 *prev*[*x*]＝NIL，元素 *x* 没有前驱，因此是线性表的表首元素或称为**表头**(***head***)。若 *next*[*x*]＝NIL，元素 *x* 没有后继，因此是线性表的表尾元素或称为**表尾**(***tail***)。属性 *head*[*L*]指向表首，*tail*[*L*]指向表尾。若 *head*[*L*]＝*tail*[*L*]＝NIL，则表是空的。

图 2-2(a)为用双向链表 *L* 表示的动态集合{1，4，9，16}。表中的每一个元素是具有关键值域和分别指向前后对象的指针域(用箭头表示)。表尾的 *next* 域和表首的 *prev* 域都是 NIL，用正斜杠表示。属性 *head*[*L*]指向表首，属性 *tail*[*L*]指向表尾。图 2-2(b)执行了将 *key*[*x*]＝25 的结点插入到 *L* 表首后，链表就具有一个关键值为 25 的对象作为表首 *head*[*L*]。图 2-2(c)接着是删除关键值为 4 的结点后的链表 *L*。

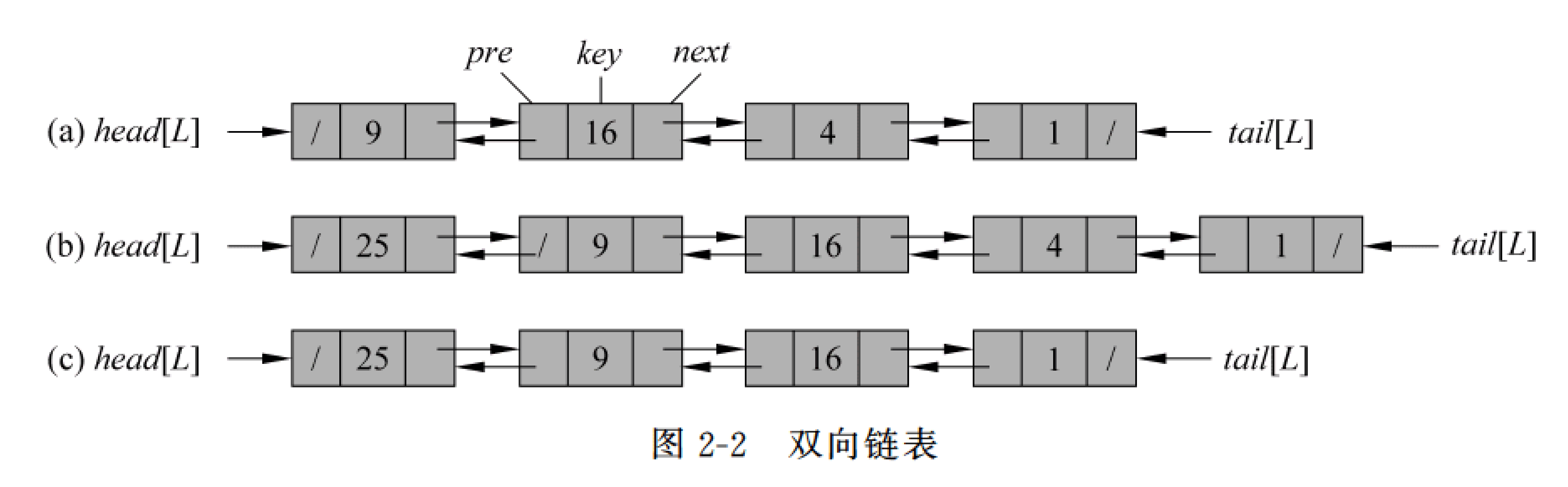


图 2-2　双向链表

在一个链表中，若表首的 *prev* 域指向表尾，而表尾的 *next* 域指向表首，称为**环形表**。此时表可视为一个由元素构成的环。这样，链表中任何一个结点都有唯一前驱和后继。

2.1.2　对链表的操作

图 2-3 为一个具有哨兵的环形双向链表。哨兵出现在表首和表尾之间。属性 *head*[*L*]不再需要，因为可以通过 *next*[*nil*[*L*]]来访问表首。同样地，可以用 *prev*[*nil*[*L*]]替代 *tail*[*L*]。图 2-3(a)为一个空链表。图 2-3(b)是由图 2-2(a)得来的链表，具有键值 9 的表首以及具有键值 1 的表尾。图 2-3(c)为执行了 LIST-INSER(*L*，*x*)后的链表，其中 *key*[*x*]＝25。新的对象称为表首。图 2-3(d)为删除键值为 1 的对象后的链表，新的表尾是键值为 4 的对象。

1. 哨兵结点

对链表的操作通常要处理链表中的某些结点。作为链表的“边界”，头结点无前驱，而尾结点无后继，对它们的处理和其他均有前驱和后继的结点会有所不同。为简化算法对边界条件的检测，为链表 *L* 添加一个哨兵结点 *nil*[*L*]：它不表示任何数据元素，但它是头结点的前驱，尾结点的后继，如图 2-3 所示。初始时，*next*[*nil*[*L*]]＝*prev*[*nil*[*L*]]＝*nil*[*L*]。添加了哨兵后，形成了一个环形链表，其中的每个结点都有各自的前驱和后继。特别地，*next*[*nil*[*L*]]表示头结点，*prev*[*nil*[*L*]]表示尾结点。这样，还可省略 *L* 的属性 *head*[*L*]和 *tail*[*L*]。

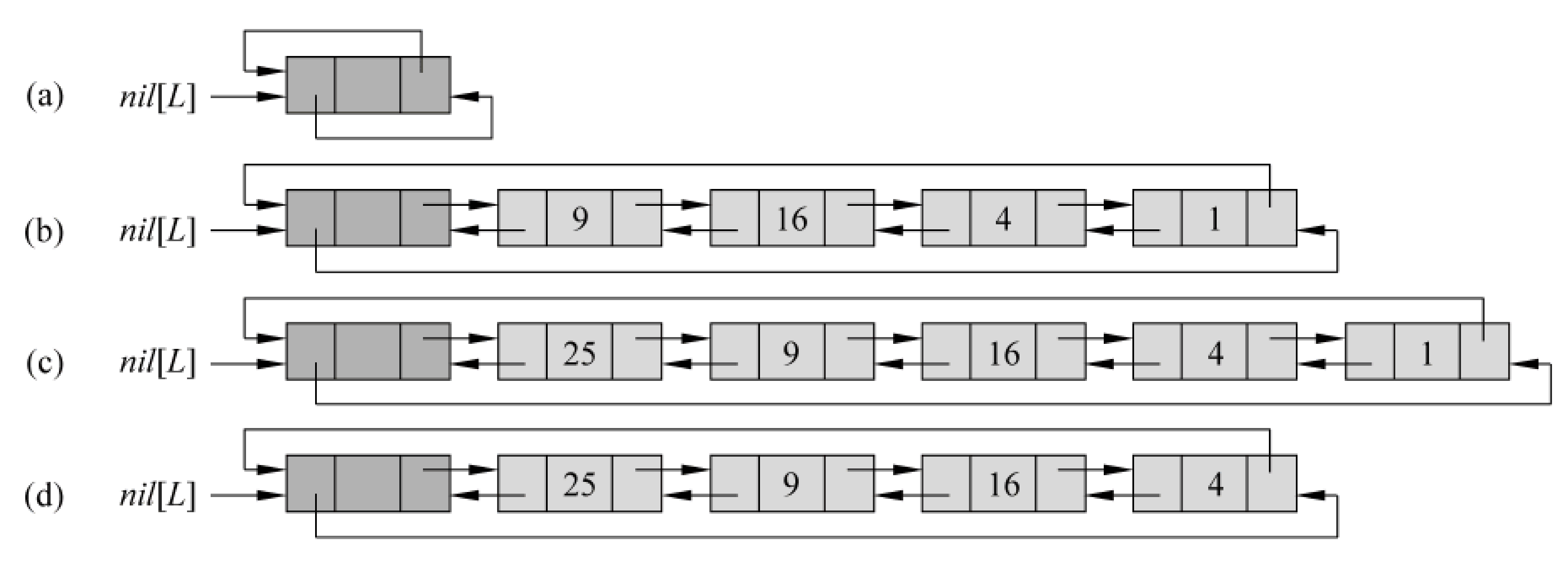


图 2-3　带有哨兵结点的双向链表

2. 对链表的扫描

过程 LIST-DISPLAY 从表头开始,逐一输出链表中每个结点的关键值域 *key*。这是典型的对线性表的扫描操作:依次对表中的每个元素进行处理。

```
LIST-DISPLAY(L)
1 x←next[nil[L]]              ▷x 指向头结点
2 while x≠nil[L]              ▷x 指向链表中的正常结点
3   do print key[x]
4      x←next[x]
```

算法 2-1　输出链表 L 的 LIST-DISPLAY 过程

其中参数 L 表示一个链表。若 L 中有 n 个结点,则过程 LIST-DISPLAY 的运行时间为$\Theta(n)$。对于双链表而言,扫描可以是“反向”的:这只要将第 1 行改为 $x \leftarrow prev[nil[L]]$,而将第 4 行改为 $x \leftarrow prev[x]$即可。

3. 在链表中查找

过程 LIST-SERACH(L,k)用简单线性搜索法在表 L 中寻找第一个值为 k 的元素,返回指向该元素的结点指针。若表中不存在值为 k 的结点,则返回 NIL。对图 2-3(a)中的链表,调用 LIST-SERACH(L,4)将返回指向第三个元素的指针,而调用 LIST-SERACH(L,7)将返回 NIL。

```
LIST-SEARCH (L,k)
x←next[nil[L]]
while x ≠nil[L] and key[x]≠k
  do x←next[x]
return x
```

算法 2-2　在双向链表 L 中查找值为 x 的元素的 LIST-SEARCH 过程

为在具有 n 个元素的链表中查找,过程 LIST-SERACH 在最坏情形下耗时 $\Theta(n)$,这是因为可能需要查遍整个链表。

4. 将元素插入到链表中

图 2-4 为将结点 x 插在结点 a 之前。由于 a 是带有哨兵的链表 L 中的结点,所以必存

在 a 的前驱。图 2-4(a)中令 $prev[a]$为 b。图 2-4(b)中令 $next[x]$为 a，$prev[x]$为 b；再令 $next[b]$和 $prev[a]$均为 x。

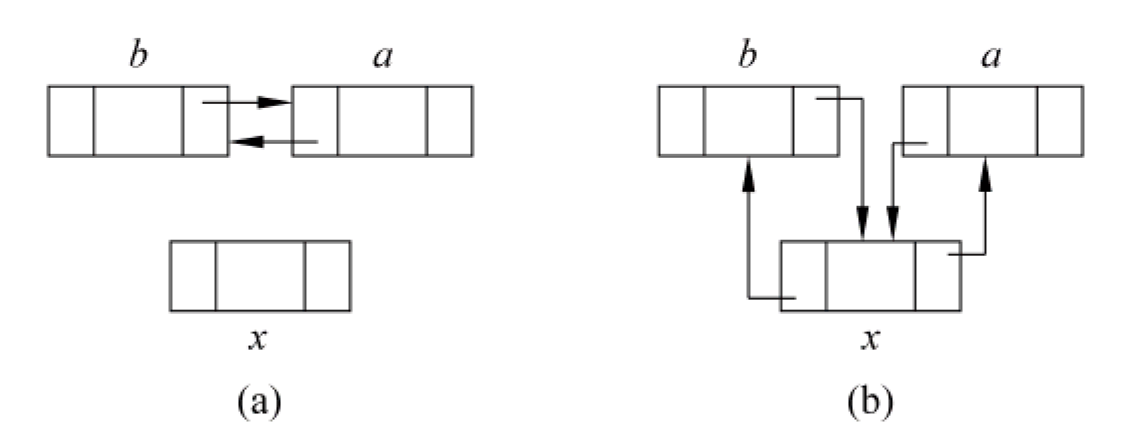


图 2-4　把元素插入到链表中

给定结点 x，其 key 域已经设置好，LIST-INSERT 过程将 x 插入到链表 L 中结点 a 的前面。LIST-INSERT 过程将 x 以如图 2-4(b)所示那样"接合"到链表结点 a 之前。

```
LIST-INSERT(L,a,x)            ▷在 L 的结点 a 之前插入新的结点 x
if x=nil[L]
   then return
b←prev[a]
prev[x]←b
next[x]←a
next[b]←prev[a]←x
```

算法 2-3　在双向链表 L 的结点 a 之前插入新结点 x 的 LIST-INSERT 过程

对一个具有 n 个元素的链表 L，LIST-INSERT 的运行时间是 $O(1)$。

5. 从链表中删除元素

过程 LIST-DELETE 将链表 L 中结点 x 从中移除。必须给定一个指向 x 的指针，然后通过修改指针将 x 从表中移除。若希望移除具有给定关键值的元素，应首先调用 LIST-SEARCH 来检索指向该元素的指针。

图 2-5 为将结点 x 从链表中删除。由于 x 是带有哨兵的链表 L 中的结点，所以 $prev[x]$和 $next[x]$都是存在的。图 2-5(a)中令 a 为 $next[x]$，b 为 $prev[x]$。图 2-5(b)中令 $next[b]$为 a，$prev[a]$为 b。

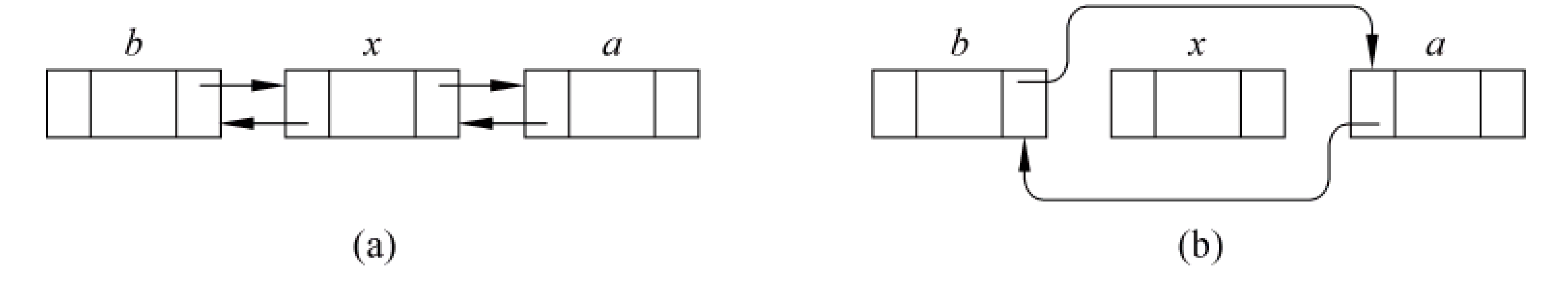


图 2-5　从链表中删除元素

```
LIST-DELETE(L,x)
if x=NIL          ▷空结点
   then return
a←next[x]
b←prev[x]
```

next[*b*]←*a*
prev[*a*]←*b*

算法 2-4 在双向链表 L 中删除结点 x 的 LIST-DELETE 过程

图 2-5 展示了如何从链表中删除一个结点。LIST-DELETE 的运行时间是 $O(1)$，但若希望删除一个具有指定关键值的结点，在最坏情形下需要 $\Theta(n)$时间，这是因为必须先调用 LIST-SERACH。

2.1.3 链表的程序实现

下面用 C 语言来实现通用链表。

1. 类型定义与函数原型声明

```
1 typedef struct ListNode{                                   /*结点类型结构体*/
2     void * key;                                            /*关键值指针*/
3     struct ListNode * prev;                                /*前驱结点指针*/
4     struct ListNode * next;                                /*后继结点指针*/
5 } ListNode;
6 ListNode * createListNode(void * d,int size);              /*创建结点*/
7 void clrListNode(ListNode * x,void( * proc)(void * ));     /*清理结点*/
8 typedef struct {                                           /*链表类型结构体*/
9     unsigned long eleSize;                                 /*元素数据存储宽度*/
10    int ( * comp)(void * ,void * );                        /*元素数据比较规则*/
11    ListNode * nil;                                        /*头指针*/
12    int n;                                                 /*元素个数*/
13 } LinkedList;
14 LinkedList * createList(unsigned long size,int( * comp)(void * ,void * ));
                                                             /*创建新链表*/
15 void clrList(LinkedList * L,void( * proc)(void * ));      /*清理链表*/
16 int listEmpty(LinkedList * );                             /*检测链表是否为空*/
17 void listTravers(LinkedList * L,void( * proc)(void * ));  /*遍历链表*/
18 ListNode * listSearch(LinkedList * L,void * e);           /*在链表中查找*/
19 void listInsert(LinkedList * L,ListNode * a,void * k);    /*在结点 a 前插入结点 k*/
20 void listDelete(LinkedList * L,ListNode * e);             /*在链表中删除结点*/
21 void listPushFront(LinkedList * L,void * k);              /*在表首插入*/
22 void listPushBack(LinkedList * L,void * k);               /*在表尾插入*/
```

程序 2-1 链表结点、链表定义及链表操作函数的声明

对程序 2-1 的说明如下。

(1) 我们的目标是实现一个能容纳任何类型数据元素的通用链表。在第 1～5 行所定义的双链表的结点数据类型 ListNode 中，将关键值域 key 定义为能指向任何类型数据的 **void** * 指针。链域 prev 和 next 分别是指向本结点的前驱结点、后继结点的指针。

(2) 第 6 行声明了用来创建结点的函数 ListNode * createListNode(**void** * d，**int**

size)，该函数用传递给它的存储在内存地址 d 处的关键值初始化结点中 key 指向的内存单元。参数 size 表示关键值的存储宽度。

(3) 由于 ListNode 的 key 域要动态分配存储关键值的空间，当某个结点从链表中删除并不再使用时，为防止内存泄漏，第 7 行声明了一个清理结点空间的函数 **void** clrListNode (ListNode * x, **void**(* proc)(**void** *))，它负责将传递给它的 x 所指向的结点的 key 域指向的内存释放掉。因为 key 还可能含有下一层动态空间域，参数 **void**(* proc)(**void** *)用来处理 x 的 key 域。

(4) 第 8～13 行定义了通用链表类型 LinkedList。它包含表示元素数据存储宽度的正整数 eleSize，表示元素数据间的比较规则的函数指针 comp，指向哨兵结点指针 nil，表中元素个数 n 等属性。

(5) 第 14 行声明的函数 LinkedList createList(**unsigned** long size, **int**(* comp)(**void** * , **void** *))用整型参数 size 初始化链表的结点数据域的存储宽度，用函数指针参数 comp 初始化链表的结点数据的比较规则，创建一个空的链表，并将指向该链表的指针作为返回值返回。第 15 行声明的函数 **void** clrList(LinkedList * L, **void** (* proc) (**void** *))负责清理链表 L 中的各结点空间，防止内存泄漏。其中，参数 **void** (* proc) (**void** *)用来处理结点的 key 域。第 16 行声明的函数 int listEmpty (LinkedList * L)检测链表 L 是否为空。

(6) 第 17～20 行声明的函数 **void** listTravers(LinkedList * L, **void** (* proc) (**void** *))、listNode * listSearch(LinkedList * L, **void** * e, **int** (* comp)(**void** * , **void** *))、void insert(LinkedList * L, listNode * a, void * k, **int** size)和 **void** listDelete (LinkedList * L, listNode * x)分别实现了算法 2-1～算法 2-4 中的过程 LIST-DISPLAY、LIST-SEARCH、LIST-INSERT 和 LIST-DELETE。

(7) 考虑到插入元素多发生在链表的两端，第 21～22 行声明的函数 listPushFront 和 listPushBack 分别执行在表首插入和在表尾插入元素。

2. 结点的常规维护

对程序 2-1 的第 1～5 行定义的链表结点类型 ListNode 的数据有两个常规维护操作：创建结点和清理结点存储空间。

```
ListNode * createListNode(void * d, int size) {          /* 创建结点 */
  ListNode * x=(ListNode * ) malloc(sizeof (ListNode));  /* 为 x 分配空间 */
  assert(x!=NULL);                                       /* 若 x 为空，中断 */
  if(d&&size){                                           /* 需要加载关键值 */
    x->key=(void * ) malloc(size);                       /* 分配空间 */
    assert(x->key!=NULL);                                /* 若 x->key 为空，中断 */
    memcpy(x->key,d,size);                               /* 加载关键值 */
    x->prev=x->next=NULL;                                /* 链域置空 */
  }
  return x;
}
void clrListNode(ListNode * x,void( * proc)(void * )) {  /* 清理结点 */
    if (x !=NULL) {
```

```
14          if (proc != NULL)                          /* 处理关键值 */
15              proc(x->key);
16          free(x->key);                              /* 释放 x->key */
17      }
18      x->next = x->prev = NULL;                      /* 链域置空 */
19}
```

程序 2-2　链表结点的常规维护函数定义

对程序 2-2 的说明如下。

(1) 第 1～11 行定义了创建链表结点的函数 ListNode * createListNode(**void** * d,**int** size)。它需要为新结点 x 分配空间(第 2 行和第 3 行),为新结点 x 中的 key 域分配空间(第 5 行和第 6 行),然后调用库函数 memcpy 把第一个参数 d 指引的宽度为第二个参数 size 的内存中的数据复制到 x 的 key 域中(第 7 行),之所以要调用 memcpy 是因为对 void * 类指针指引的内存不能直接赋值。函数 memcpy 的原型为

```
void * memcpy(void * dest, const void * src, unsigned long size)
```

其功能是将以 src 指向的内存地址开始 size 个字节内的数据复制到以 dest 指向的 size 个字节内,如图 2-6 所示。函数 memcpy 的原型声明于头文件<string.h>中。最后返回 x。

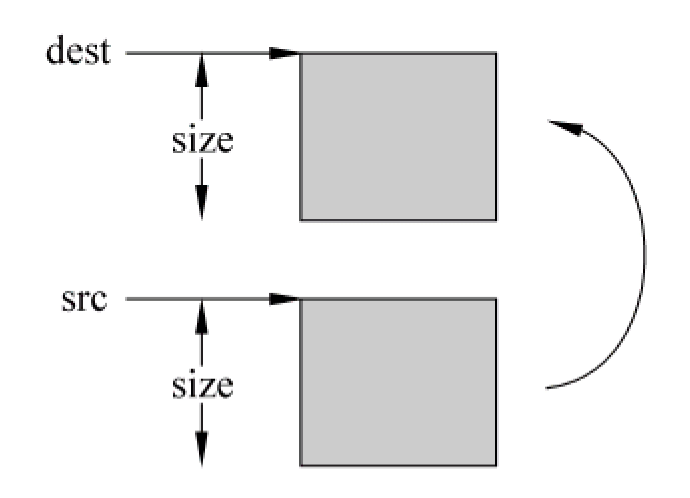


图 2-6　memcpy(void * dest, const void * src, unsigned long size)示意

(2) 由于每个结点的 key 域指向动态内存空间,所以第 12～19 行定义的清理结点空间函数 **void** clrListNode(ListNode * x,**void**(* proc)(**void** *))对 x 的 key 域做空间释放操作(第 16 行)。其中,第 14～15 行利用参数 proc 对结点 x 的 key 域做释放前的处理。

3. 链表常规维护

对程序 2-1 的第 8～13 行定义的链表类型 LinkedList 数据的常规维护包括创建链表、清理链表存储空间及判断表空。

```
LinkedList * createList(unsigned long size, int( * comp)(void * , void * )) {   /* 创建新链表 */
    LinkedList * L = (LinkedList * ) malloc(sizeof (LinkedList));             /* 分配空间 */
    assert(L != NULL);                                        /* 若 L 为空,中断 */
    L->nil = createListNode(NULL, 0);                         /* 将 nil 的关键值置空 */
    L->nil->prev = L->nil->next = L->nil;                     /* 将 nil 的链域指向自身 */
    L->n = 0;                                                 /* 结点数置为 0 */
    L->eleSize = size;                                        /* 设置元素数据存储宽度 */
    L->comp = comp;                                           /* 设置元素数据比较规则 */
```

```
9      return L;
10 }
11 void clrList(LinkedList * L,void( * proc)(void * )) {          /* 清理链表 */
12     ListNode * x=L->nil->next;                                  /* x 指向头结点 */
13     while (x!=L->nil) {                                         /* x 非 nil */
14         listDelete(L,x);                                        /* 删除 x */
15         clrListNode(x,proc);                                    /* 清理 x */
16         free(x);                                                /* 释放 x 的空间 */
17         x=L->nil->next;                                         /* x 指向头结点 */
18     }
19     free(L->nil);                                               /* 释放哨兵结点指针空间 */
20     L->comp=NULL;
21 }
22 int listEmpty(LinkedList * L) {                                 /* 检测链表是否为空 */
23     return L->nil->next==L->nil;
24 }
```

程序 2-3　链表常规维护函数的定义

对程序 2-3 的说明如下。

(1) 第 1～10 行定义了创建空表的函数 LinkedList * createList(**unsigned long** size,**int** (* comp)(**void** * ,**void** *))。该函数为指向新的链表的指针 L 分配空间(第 2 行和第 3 行),为 L 创建哨兵结点 nil(第 4 行),将 L 置空(第 5 行和第 6 行,空表的特征是哨兵结点 nil 的 prev 域和 next 域均指向自身且元素个数 n 为 0)。用整型参数 size 初始化元素存储宽度(第 7 行),用函数指针参数 comp 初始化链表元素的比较规则(第 8 行)。并将 L 作为返回值(第 9 行)。

(2) 第 11～21 行定义了清理链表空间的函数 **void** clrList(LinkedList * L),它通过第 13～18 行的 **while** 循环依次将当前表头结点从表 L 中摘除(第 14 行),然后清理该结点空间(第 15 行和第 16 行),直至表空。最后释放 L 的哨兵结点 nil 的空间(第 19 行)。

(3) 定义在第 22～24 行的函数 **int** listEmpty(LinkedList * L),用表 L 中的头结点(哨兵结点 nil->next)是否为哨兵结点 nil 来判断该表是否为空。

4. 链表字典操作

下列 C 函数实现算法 2-1～算法 2-4 中关于链表的字典操作过程 LIST-DISPLAY、LIST-SEARCH、LIST-INSERT 和 LIST-DELETE 等。利用 LIST-INSERT 过程的实现函数还可定义常用的在表首插入元素和在表尾插入元素的函数。

```
void listTravers(LinkedList * L,void( * proc)(void * )) {          /* 输出链表 */
  ListNode * x=L->nil->next;
  while (x !=L->nil) {
       proc(x->key);
       x=x->next;
  }
}
```

```
8 ListNode * listSearch(LinkedList * L, void * e) {                /* 在链表中查找 */
9     int (* comp)(void *, void *)=L->comp;
10    ListNode * x=L->nil->next;                                   /* x 指向头结点 */
11    while (x !=L->nil && comp(x->key,e) !=0)                     /* x 非 nil 且关键值非 e */
12        x=x->next;                                               /* x 后移一个结点 */
13    return x;
14 }
15 void listInsert(LinkedList * L, ListNode * a, void * k) {       /* 在 L 的 a 之前插入元素 k */
16    int size=L->eleSize;
17    ListNode * x=createListNode(k,size), * b=a->prev;
18    L->n++;
19    x->next=a;
20    x->prev=b;
21    a->prev=b->next=x;
22 }
23 void listDelete(LinkedList * L, ListNode * x) {                 /* 在链表中删除结点 */
24    if (x ==L->nil)
25        return;
26    ListNode * a=x->next,
27     * b=x->prev;
28    b->next=a;
29    a->prev=b;
30    L->n--;
31 }
32 void listPushFront(LinkedList * L, void * k){                   /* 在表首插入 */
33    listInsert(L,L->nil->next,k);
34 }
35 void listPushBack(LinkedList * L, void * k){                    /* 在表尾插入 */
36    listInsert(L,L->nil,k);
37 }
```

程序 2-4　定义链表的字典操作的 C 函数

对程序 2-4 的说明如下。

(1) 第 1～7 行定义的函数 **void** listTravers(LinkedList * L,**void**(* proc)(**void** *))实现的是算法 2-1 的 LIST-DISPLAY 过程,不过并不限于输出链表的每个结点,而是通过传递函数指针参数 proc 处理结点的 key 域(第 4 行)。

(2) 第 8～14 行定义的函数 ListNode * listSearch(LinkedList * L,**void** * e)实现的是算法 2-2 的 LIST-SEARCH 过程。和算法过程一样,该函数有两个参数:链表 L 和待查数据 e。之所以参数如此简洁,是因为把元素数据比较规则设置为链表的属性。第 9 行将 L 的 comp 属性赋值给函数指针变量 comp,这是为了使代码更简练。

(3) 第 15～22 行定义的函数 **void** listInsert(LinkedList * L,ListNode * a,**void** * k)实现的是算法 2-3 中的 LIST-INSERT 过程,在链表 L 的结点 a 之前插入关键值为 k 的结点。程序代码与 LIST-INSERT 的伪代码几乎一致,读者可对照阅读理解。

第 32～34 行定义的函数 **void** listPushFront(LinkedList * L,**void** * k)调用函数 listInsert,将新结点插入到表头。

第 35～37 行定义的函数 **void** listPushBack(LinkedList * L,**void** * k)也是利用 listInsert 将新结点插入到表尾。

(4) 第 23～31 行定义的函数 **void** listDelete(LinkedList * L,ListNode * x)实现的是算法 2-4 中的 LIST-DELETE 过程,程序代码与 LIST-DELETE 过程的伪代码也十分相近。

为便于重用,把程序 2-1 保存为文件夹 datastructure 中的头文件 list.h,把程序 2-2～程序 2-4 保存到文件夹 datastructure 中的源文件 list.c 内。

2.1.4 链表应用

数据规范问题

问题描述

在很多信息处理的应用中,收集到的原始数据往往有一些重复项,因此需要做预处理:剔除数据集合中的重复项并按一定的顺序排好序。例如,原始数据为整数序列<5,3,7,3,4,8,7,9,2,8,5,1>,处理后的序列为<1,2,3,4,5,7,8,9>。

输入数据

输入文件 inputdata.txt 包含若干行,每一行表示一个案例。每一行包含若干数据项,数据项之间用一个空格隔开。每一行的第一个数据项是一个字符,值为'i'、'c'、'f'、's'之一,分别意为本案例中数据的类型为整型、字符型、浮点型或字符串型。行中剩余各项为一组类型相同的数据。

输出数据

对应输入文件中的每一个案例,向输出文件 outputdata.txt 输出一行处理后的无重复项且按升序排列的数据,数据项之间用一个空格隔开。

输入样例

```
s Wang Li Xie Wu Li Huang Wang
i 5 3 7 3 4 8 7 9 2 8 5 1
f 3 4.1 3.7 5.2 4.1 3.7 1.5 3.0
c a c p l c b d p m m o x
```

输出样例

```
Huang Li Wang Wu Xie
1 2 3 4 5 7 8 9
1.500000 3.000000 3.700000 4.100000 5.200000
a b c d l m o p x
```

1. 算法描述

解决数据规范问题需要注意输入文件的格式:由若干行组成,每行表示一个任务,其中包括的信息为数据类型和同种类型的若干数据。与搜索引擎问题相仿,本问题的关键点在

于各个任务要处理的数据类型不尽相同。我们的想法是对每个搜索任务读取一行数据,从中析取第一个字符确定任务中数据的类型,处理过程维护一个线性表 A,存储任务中不重复的数据,初始时 $A=\varnothing$。接下来对任务中的每一个数据调用 LINEAR-SEARCH 检测是否在 A 中出现,若否则将其添加到 A 的合适的位置,使其有序。最后将保存在 A 中的数据转换成一行文本写到输出文件中。算法伪代码过程如下。

```
NORMALIZE()
1 打开输入文件 inputdata.txt,其为 f1
2 打开输出文件 outputdata.txt,其为 f2
3 while not end of f1
4   do s←从 f1 中读取一行
5      type←从 s 中读取第一个字符
6      s←s 中去掉第一个字符后的部分
7      t←PROCCESS(type,s)
8      将串 t 作为一行写入文件 f2
9 关闭 f1
10 关闭 f2
```

算法 2-5　解决数据规范问题的算法伪代码过程

其中第 7 行对过程 PROCCESS($type$,s)的调用就是对含在 s 中的各项 $type$ 类型数据进行处理,剔除其中的重复项,剩下的元素按升序表示为一个字符串,并返回。伪代码描述如下。

```
PROCCESS(type,s)
创建链表 L
t←空串
for s 中每项 type 类型数据 x
  do if LIST-SERACH (L,x)=nil[L]
     then 将 x 插入到 L 的合适位置使得 L 有序
for L 中每个结点 e
  do 将 e 中数据追加到串 t 的尾部
return t
```

算法 2-6　按数据类型处理数据案例的过程

2. 程序实现

1) 串输出流

为了实现算法中将一个链表中的数据逐项追加到一个串的尾部,需要实现类似于第 1 章中开发的串输入流 StrInputStream 的串输出流 StrOutputStream。定义如下。

```
typedef struct{
        char *  begin;                          /*输出流首地址*/
        char *  current;                        /*输出流当前写入位置*/
        int length;                             /*输出流长度*/
}StrOutputStream;                               /*串输出流*/
void initStrOutputStream(StrOutputStream *,int);  /*初始化输出流*/
```

```
7 void freeStrOutputStream(StrOutputStream *);               /* 清理输出流空间 */
8 int sosFull(StrOutputStream *);                            /* 检测输出流是否满 */
9 void sosFresh(StrOutputStream *);                          /* 输入流还原初始设置 */
10 int writeInt(StrOutputStream *,int);                      /* 向串输出流写入整型数据 */
11 int writeDouble(StrOutputStream *,double);                /* 向串输出流写入浮点型数据 */
12 int writeChar(StrOutputStream *,char);                    /* 向串输出流写入字符型数据 */
13 int writeString(StrOutputStream *,char *);                /* 向串输出流写入字符串数据 */
```

程序 2-5　串输出流类型的定义以及串输出流操作函数的声明

对程序 2-5 的说明如下。

(1) 第 1～5 行定义的是串输出流 StrOutputStream。它含有两个字符型指针属性：begin 和 current，分别指向串流首和当前读取数据的位置。此外，还有一个表示串输出流空间大小的整型属性 length。

(2) 第 6～9 行声明了用来初始化串输出流、清理串输出流空间、检测串输出流是否满、清空串输出流中数据的常规维护函数 initStrOutputStream、freeStrOutputStream、sosFull、sosFresh。

(3) 第 10～13 行声明了用来向串输出流写入整型数据、浮点型数据、字符型数据、字符串数据的函数 writeInt、writeDouble、writeChar、writeString。

2) **串输出流的常规维护**

```
1 void initStrOutputStream(StrOutputStream * sout,int size){   /* 初始化输出流 */
2     if(sout->begin=(char *)malloc(size * sizeof(char))){     /* 分配串输出流空间 */
3         sout->length=size;
4         memset(sout->begin,0,size * sizeof(char));
5         sout->current=sout->begin;
6     }
7 }
8 void freeStrOutputStream(StrOutputStream * sout){             /* 清理输出流空间 */
9     sout->length=0;
10    sout->current=NULL;
11    if(sout->begin)
12        free(sout->begin);
13 }
14 void sosFresh(StrOutputStream * sout){
15    sout->current=sout->begin;
16 }
17 int sosFull(StrOutputStream * sout){                         /* 检测输出流空间是否满 */
18    return sout->current-sout->begin>=sout->length;
19 }
```

程序 2-6　串输出流的常规维护函数

对程序 2-6 的说明如下。

(1) 第 1～7 行定义的函数 initStrOutputStream 通过为串输出流 sout 的 begin 指针分配空间并将写入位置 current 赋值为 begin，串空间长度 length 赋值为 size，初始化输出流

sout。

(2) 第 8～13 行定义的函数 freeStrOutputStream 负责释放不再使用的串输出流 sout 的 begin 指针的动态存储空间,以防内存泄漏。

(3) 第 14～16 行定义的函数 sosFresh 通过将串输出流 sout 的当前写指针 current 赋值为 begin,清空输出流中的数据,以备重新使用。

(4) 第 17～19 行定义的函数 sosFull 通过检测输出流中的 current 指针是否指向空间的末尾来判断输出流是否满。

3) 向串输出流写数据

可以向串输出流写入各种类型的数据,为节省篇幅,此处仅列出向串输出流写入整型数据的 C 函数 writeInt,向输出流写入其他基本类型数据的函数可打开相应的源文件研读。

```
1 int writeInt(StrOutputStream * sout,int x){          /* 向输出流写入整型数据 */
2     if(sosFull(sout))                                  /* 流空间已满 */
3         return 0;
4     if(sout->current==sout->begin)                    /* 写入的是第一个数据 */
5         sprintf(sout->current,"%d",x);
6     else
7         sprintf(sout->current,"%c%d",' ',x);
8     sout->current+=strlen(sout->current);             /* 刷新当前写入位置 */
9     return 1;
10 }
```

程序 2-7　向串输出流写入整型数据的 C 函数

对程序 2-7 的说明如下。

(1) 该函数有两个参数,串输出流指针 sout 和要向 sout 写入的整型数据 x。写入成功,返回 1,否则返回 0。

(2) 第 2～3 行检测输出流空间是否满。第 4～7 行的 **if…else** 语句根据写入的数据是否为第一个,决定输出数据前是否加入一个空格,这样写入输出流中的数据以空格作为分隔符。第 8 行为下一次输出数据刷新写入位置 current。以此方式,可以连续地向一个串写入多项数据。

为便于代码重用,将程序 2-5 添加到 utility 文件夹中的头文件 strstream. h 中,将程序 2-6、程序 2-7 添加到同一文件夹的源文件 strstream. c 中。

利用程序 2-1～程序 2-7,可用 C 语言实现解决"数据规范问题"的程序。

4) 主调函数

由于各个案例的数据类型有所不同,所以尽管处理流程近似,却不能像算法那样用一个函数统一处理而需要对不同数据类型分别编写处理函数。然后通过一个主调函数,按不同的数据类型调用分类型处理函数各种类型的案例数据。

```
char * proccess(char type,char * s){       /* 按 type 表示的数据类型处理存储在 s 中的数据 */
    switch(type){
        case 'i': return intProccess(s);
```

```
4           case 'c': return charProccess(s);
5           case 'f': return doubleProccess(s);
6           case 's': return stringProccess(s);
7       }
8       freeStrOutputStream(&ssout);
9 }
10 int main(int argc,char** argv) {
11     FILE *f1=fopen("chap02/normalize/inputdata.txt","r"),      /*打开输入数据文件*/
12          *f2=fopen("chap02/normalize/outputdata.txt","w");     /*打开输出数据文件*/
13     char s[250];
14     assert(f1&&f2);
15     while(!feof(f1)){                                         /*如果还有输入数据*/
16         char type,*t;
17         fgets(s,250,f1);                                      /*从中读取一行 s*/
18         type=*s;
19         t=proccess(type,s+2);
20         fputs(t,f2);                          /*将处理好的数据作为一行写入输出文件*/
21     }
22     fclose(f1);fclose(f2);                                    /*关闭数据文件*/
23     return (EXIT_SUCCESS);
24 }
```

程序 2-8　解决“规范数据”问题的 C 程序

对程序 2-8 的说明如下。

(1) 第 10～24 行定义的 main 函数实现的是算法 2-5 的 NORMALIZE 过程。程序代码与算法的伪代码非常相近，在此不再赘述。

(2) 第 1～9 行定义的函数 **char** * proccess(**char** type, **char** * s)实现算法 2-6 的 PROCCESS 过程，第 2～7 行的 **switch** 语句根据参数 type 表示的数据类型调用对不同类型的处理函数读取含在串 s 中的数据进行处理，处理后的数据表示在一个字符串中作为函数值返回。

5) **整型数据处理**

虽然对各种类型的数据处理步骤几乎一致，但 C 语言不能将类型参数化，所以需要对每一种类型的数据写一套过程。为节省篇幅，此处仅列出对整型数据的处理过程，对浮点型数据、字符型数据和字符串数据的处理过程请打开相应文件比较研读。

```
char *intProccess(char *s){                              /*整型数据处理函数*/
    StrInputStream ssin;                                 /*创建串输入流*/
    LinkedList *L=createList(sizeof(int),intGreater);    /*创建整型链表 L*/
    int x;
    initStrInputStream(&ssin,s);                         /*用 s 初始化串输入流*/
    initStrOutputStream(&ssout,250);                     /*初始化串输出流*/
    while(!sisEof(&ssin)){                               /*处理输入流中的数据*/
        readInt(&ssin,&x);                               /*从输入流中读取一项数据 x*/
        if(listSearch(L,&x)==L->nil){                    /*若 x 没有出现在表 L 中*/
```

```
10          ListNode *a=L->nil->next;                      /*a从头结点开始*/
11          while(a!=L->nil&&*((int*)(a->key))<x)          /*找到插入位置*/
12              a=a->next;
13          listInsert(L,a,&x);                            /*在a之前插入*/
14      }
15  }
16  listTraverse(L,putInt);                   /*将链表中数据逐项写到串输入流中*/
17  clrList(L,NULL);
18  writeString(&ssout,'\n');                                  /*构成一行*/
19  return ssout.begin;
20 }
```

程序 2-9 处理整型数据的 C 函数

对程序 2-9 的说明如下。

(1) 第 3 行创建一个链表 L,第 5 行用传递给它的参数 s 创建一个串输入流 ssin,第 6 行初始化串输出流 ssout 缓冲区长度为 250B。第7～15 行的 **while** 循环逐一读取 ssin 中的数据项到 x 中(第 8 行),调用 listSearch 函数检测 x 是否在 L 中存在(第 9 行),若 x 未出现在 L 中,第 11～12 行通过一个 **while** 循环确定关键值为 x 的结点的插入位置(由 a 指引)。第 13 行调用函数 listInsert,将 x 插入到 L 中 a 之前。这样,L 中存储了 ssin 中所有各不相同的数据项且有序。第 16 行调用 listTraverse 函数,用传递给它的 putInt 将 L 中的元素逐一写到 ssout 中。第 17 行调用 clrList 函数清理链表 L 的存储空间,防止内存泄漏。第 19 行将写到 ssout 中的数据构成的串作为返回值返回。

(2) 由于 listTraverse 函数的第二个参数是 **void**(*proc)(**void***)的函数指针,仅含一个指针参数,为能正确地将每个结点的数据写到串输出流中,需要定义一个全局性的串输出流 ssout,并定义类型为 **void**(*proc)(**void***)的函数将指定类型数据写入 ssout。这些工作,都添加到第 1 章创建的 strstream.h 和 strstream.c 中。

```
⋮
StrOutputStream ssout;/*全局串输出流*/
void putInt(int*);
void putChar(char*);
void putDouble(double*);
void putString(char*);
⋮
```

程序 2-10 加入到头文件 strstream.h 中的全局串输出流 sout 的定义及用于向 sout 写入基本数据类型数据的函数声明

```
⋮
void putInt(int *data){
    writeInt(&ssout,*data);
}
void putChar(char *data){
    writeChar(&ssout,*data);
}
```

```
void putDouble(double * data){
    writeDouble(&ssout, * data);
}
void putString(char * data){
    writeString(&ssout,data);
}
⋮
```

程序 2-11　加入到源文件 strstream.c 中的用于向全局串输出流 sout 写入基本数据类型数据的函数定义

程序 2-8 和程序 2-9 保存在文件夹 chap02\normalize 的源文件 normalize.c 中。

2.2　栈

栈和队列是用线性表表示的动态集合，用 DELETE 操作从这些集合中删除的元素是预先指定的。在**栈**中，从集合中删除的元素是最近才插入其中的，栈实现的是**后进先出**（**LIFO**）的策略。类似地，在**队列**中，删除的元素总是集合中留驻时间最长的，队列实现的是**先进先出**（**FIFO**）的策略。有若干种在计算机上实现栈和队列的有效方法。本节中，将说明如何利用一个链表来实现它们。

2.2.1　栈的概念及其链表实现

对栈进行的 INSERT 操作 PUSH 常称为压栈，而对不带任何参数的 DELETE 操作 POP，常称为弹出。这些名称取自实际上的栈，如餐馆里使用的带有弹簧的盘碟栈。盘子从中弹出的顺序与盘叠压入栈的顺序是相反的，所以只有栈顶的盘子才是可取的。

如图 2-7 所示，可以用一个链表来实现一个栈 S。S 有两个属性，一个是链表 $L[S]$，另一个属性是栈顶 $top[S]$，它指向最近才插入的元素（表头 $head$）。该栈由元素 $head[L[S]]$（$=top[S]$）$\cdots tail[L[S]]$组成，其中 $head[L[S]]=top[S]$是栈顶，而 $tail[L[S]]$是栈底。

当 $top[S]=$NIL 时，栈就不含有元素而成为**空**的了。可以用询问操作 STACK-EMPTY 来检测栈是否为空。若一个空的栈执行弹出操作，称其为**下溢**，这通常是一个错误。利用链表实现的栈，没有**上溢**，即栈空间满时被执行压栈操作的情形，这是因为链表空间是通过结点的增删动态管理的。

利用链表的操作，栈的各种操作都可简单地加以实现。仍然使用带有哨兵的链表作为栈 S 的属性 L。

```
STACK-EMPTY(S)
if top[S]=nil[L[S]]
    then return TRUE
    else return FALSE
PUSH(S,x)
```

```
创建一个以 x 为关键值的结点 x1
LIST-INSERT(L[S],next[nil[L[S]]],x1)
top[S]←x1
POP(S)
if STACK-EMPTY(S)
   then error "underflow"
   else x←top[S]
      top[S]←next[x]
      LIST-DELETE(L[S],x)
return x
```

算法 2-7　基于链表的栈操作算法

图 2-7 说明了变动性操作的效果。每个操作的耗时都是 $O(1)$。

图 2-7(a)中,栈 S 有 4 个元素,栈顶元素是 9。图 2-7(b)为调用 PUSH(S,17)和 PUSH(S,3)后的栈。图 2-7(c)为调用 POP(S)后的栈,POP(S)返回元素 3,该元素是最近才压入栈的。栈顶现在是元素 17。

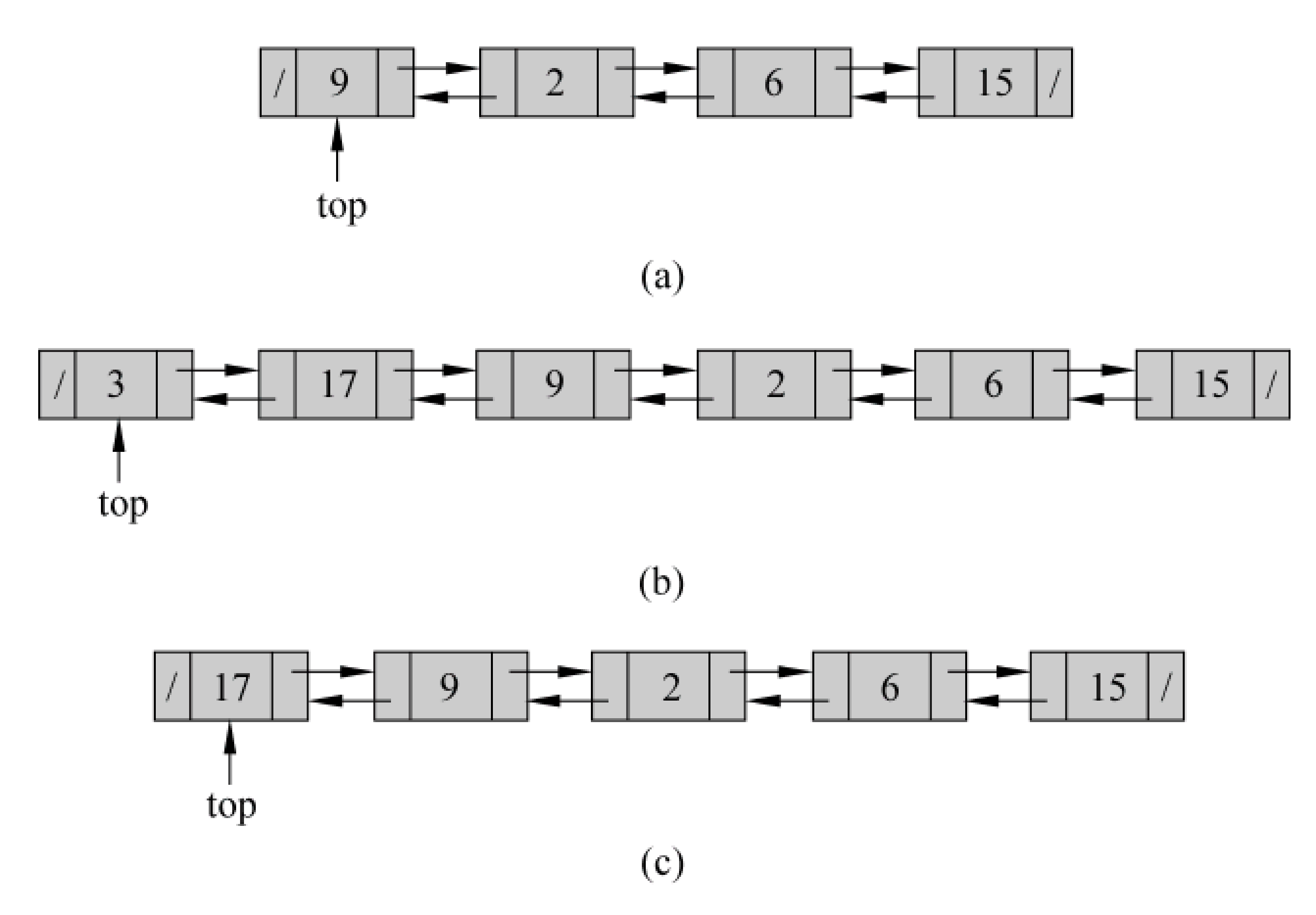


图 2-7　栈 S 的链表实现

2.2.2　栈的程序实现

在 C 语言中,程序员需要亲自实现栈数据结构。利用 2.2.1 节所实现的带有哨兵结点的链表结构 LinkedList 作为栈中元素的存储载体,将栈定义为下列结构。

```
typedef struct{                                      /* 栈类型 */
    LinkedList  * L;                                 /* 链表属性 */
    ListNode  * top;                                 /* 栈顶属性 */
}Stack;
Stack *  createStack(unsigned long size);            /* 创建空栈 */
void clrStack(Stack  * S,void( * proc)(void * ));    /* 清理栈空间 */
```

```
7 int stackEmpty(Stack *  S);                    /* 检测栈空 */
8 void push(Stack  * S,void  * k);              /* 压栈操作 */
9 ListNode  * pop(Stack  * S);                   /* 弹出操作 */
```

程序 2-12　基于链表的栈类型定义及栈操作函数的声明

对程序 2-12 的说明如下。

(1) 第 1～4 行定义栈类型 Stack。Stack 有两个属性,一个是作为元素载体的链表 L,另一个是指向栈顶元素的指针 top。其中,链表结点类型 ListNode 和链表的类型 LinkedList 及其操作函数已在 2.2.1 节加以定义。代码在文件夹 data structure 中的头文件 list.h 和源文件 list.c 中,读者可打开研读。

(2) 第 5～6 行声明了两个关于栈的常规维护函数,包括第 5 行的 createStack 函数创建一个新的栈;第 6 行的 clrStack 函数在栈使用完毕清理空间。

(3) 第 7～9 行声明了实现算法 2-7 中各过程的函数。第 7 行的 stackEmpty 函数实现算法过程 STACK-EMPTY 检测栈是否为空;第 8 行的 push 函数实现压栈算法过程 PUSH;第 9 行的 pop 函数实现弹栈算法过程 POP。

为便于代码重用,将程序 2-12 存储为 data structure 文件夹内的头文件 stack.h。并在下列的 stack.cpp 源文件中定义上述的栈操作函数。

```
#include<assert.h>
#include "stack.h"
Stack *  createStack(unsigned long size){                 /* 创建空栈 */
    Stack  * S=(Stack * )malloc(sizeof(Stack));          /* 分配空间 */
    assert(S!=NULL);                                      /* 保证空间分配成功 */
    S->L=createList(size,NULL);                           /* 创建链表属性 */
    S->top=S->L->nil;                                     /* 初始化栈顶 */
    return S;
}
void clrStack(Stack  * S,void( * proc)(void * )){
    S->top=NULL;                                          /* 栈顶指针置空 */
    clrList(S->L,proc);                                   /* 清理链表空间 */
    free(S->L);
}
int stackEmpty(Stack *  S){                               /* 检测栈空 */
    return S->top==S->L->nil;
}
void push(Stack  * S,void  * k){                          /* 压栈操作 */
    listPushFront(S->L,k);                                /* 在链表尾插入 */
    S->top=S->L->nil->next;                               /* 设置新的栈顶 */
}
ListNode  * pop(Stack  * S){                              /* 弹出操作 */
    assert(!stackEmpty(S));                               /* 检测下溢 */
    ListNode  * x=S->top;                                 /* x 设置为栈顶 */
```

```
25    S->top=x->next;                          /* 设置新的栈顶 */
26    listDelete(S->L,x);                      /* 将栈顶从链表尾部摘除 */
27    return x;
28 }
```

程序 2-13　定义栈操作的 C 函数

对程序 2-13 的说明如下。

(1) 第 3～9 行定义的函数 createStack 创建一个新的空栈。第 4 行为栈分配空间,第 6 行调用 createList 函数传递参数 size 和 NULL 设置链表属性 L。注意,程序 2-3 中定义的函数 createList 的两个参数的意义:前者 size 表示结点的数据存储宽度,后者 comp 表示结点数据大小比较的规则。由于栈中元素无须比较大小,所以第 2 个参数传递 NULL。最后第 7 行将栈顶指针置空(注意所使用的是带有哨兵结点的链表)。

(2) 第 10～14 行定义的函数 clrStack 清理 proc 作为第 1 个参数传递给它的栈 S 的存储空间。由于栈的 L 链表属性拥有动态空间,所以利用传递给它的链表结点空间处理函数指针参数 proc,第 12 行调用 clrList 函数清理 L 的存储空间,第 13 行最终释放 L 的空间。

(3) 第 15～17 行、第 18～21 行以及第 22～28 行定义的 stackEmpty、push 和 pop 函数实现算法 2-7 中的栈操作过程 STACK-EMPTY、STACK-PUSH 和 STACK-POP。实现代码与算法伪代码十分接近,读者可自行比较。

为便于代码重用,特将程序 2-13 的代码存储为文件夹 datastructure 中的源文件 stack.c。

2.2.3　栈的应用

现在用以上开发的栈结构解决下列应用问题。

Eventually periodic sequence

Given a function f: $0..N \rightarrow 0..N$ for a non-negative N and a non-negative integer $n \leqslant N$. One can construct an infinite sequence $F=f^1(n), f^2(n), \cdots, f^k(n), \cdots$, where $f^k(n)$ is defined recursively as follows: $f^1(n)=f(n)$ and $f^{k+1}(n)=f(f^k(n))$.

It is easy to see that each such sequence F is eventually periodic, that is periodic from some point onwards, e.g. 1,2,7,5,4,6,5,4,6,5,4,6…. Given non-negative integer $N \leqslant$ 11000000, $n \leqslant N$ and f, you are to compute the period of sequence F.

Each line of input contains N, x and the description of f in postfix notation, also known as Reverse Polish Notation (RPN). The operands are either unsigned integer constants or N or the variable x. Only binary operands are allowed: + (addition), * (multiplication) and % (modulo, i. e. remainder of integer division). Operands and operators are separated by white space. The operand % occurs exactly once in a function and it is the last (rightmost, or topmost if you wish) operator and its second operand is always N whose value is read from input. The following function:

$$2x * 7 + N\%$$

is the RPN rendition of the more familiar infix $(2*x+7)\%N$. All input lines are shorter than 100 characters. The last line of input has N equal 0 and should not be processed.

For each line of input, output one line with one integer number, the period of F corresponding to the data given in the input line.

Sample input

```
10 1 x N %
11 1 x x 1+ * N %
1728 1 x x 1+ * x 2+ * N %
1728 1 x x 1+x 2+ * * N %
100003 1 x x 123+ * x 12345+ * N %
0 0 0 N %
```

Output for sample input

```
1
3
6
6
369
```

1. 问题描述与分析

给定函数 $f(x)$：$\{0..N\}\to\{0..N\}$。对任一 $0\leqslant n\leqslant N$，可得到序列：

$$F=\{f(n),f^2(n),\cdots,f^k(n),\cdots\}$$

其中，$f^{k+1}(x)=f(f^k(x))$。显然，F 是一个周期循环序列。

输入文件由若干行组成，每一行的前两项为 N 和 n，余下部分是一个由逆波兰表达式(Reverse Polish Notation，RPN)给出的函数 $f(x)$ 的定义式。其中的运算对象包括作为运算数的带符号常量、N 和变量 x 以及运算符+、*、%。我们的任务是对每一行指定的 x 和 N 值以及函数 $f(x)$ 的表达式，计算序列 F，输出该序列的最小重复周期。

解决这个问题有两个关键点：首先，如何根据表示为串的逆波兰式计算函数值。其次，如何找到序列 $f(x),f^2(x),\cdots,f^k(x),\cdots$ 中的最小周期。对于前者，可以在分析逆波兰式时，利用一个栈 S 计算出表达式的值。例如，对输入样例中的串"$x\ x\ 1+\ *\ N\%$"，图 2-8 展示了计算的过程。

图 2-8(a)分析到式中第 1 个运算数 x，将其压入栈 S。图 2-8(b)分析到第 2 个运算数 x，压入栈 S。图 2-8(c)分析到第 3 个运算数 1，将其压入栈 S。图 2-8(d)分析到运算符+，弹出 S 中的两个运算数 1 和 x，相加后压入栈 S。图 2-8(e)分析到运算符 *，弹出 S 中的两个运算数 $x+1$ 和 x，相乘后压入 S。图 2-8(f)分析到运算数 N，将其压栈。图 2-8(g)分析到运算符%，弹出 S 中的两个运算数 N 和 $x*(x+1)$，计算后将所得结果 $x*(x+1)\%N$ 压栈。

对计算得到的结果，存储在 x 中，作为计算序列 $f(x),f^2(x),\cdots,f^k(x),\cdots$中的下一个元素的函数自变量用。序列 $f(x),f^2(x),\cdots,f^k(x),\cdots$可以用一个数组 $f[1..N]$存储，每计算出一个 $f^k(x)$，就保存在 $f[k]$中。并且判断 $f[k]$是否与 $f[1]\sim f[k-1]$中的某一个 $f[i]$相等。若是，则 $k-i$ 就是该序列的最小周期。否则，计算下一个 $f^{k+1}(x)$。

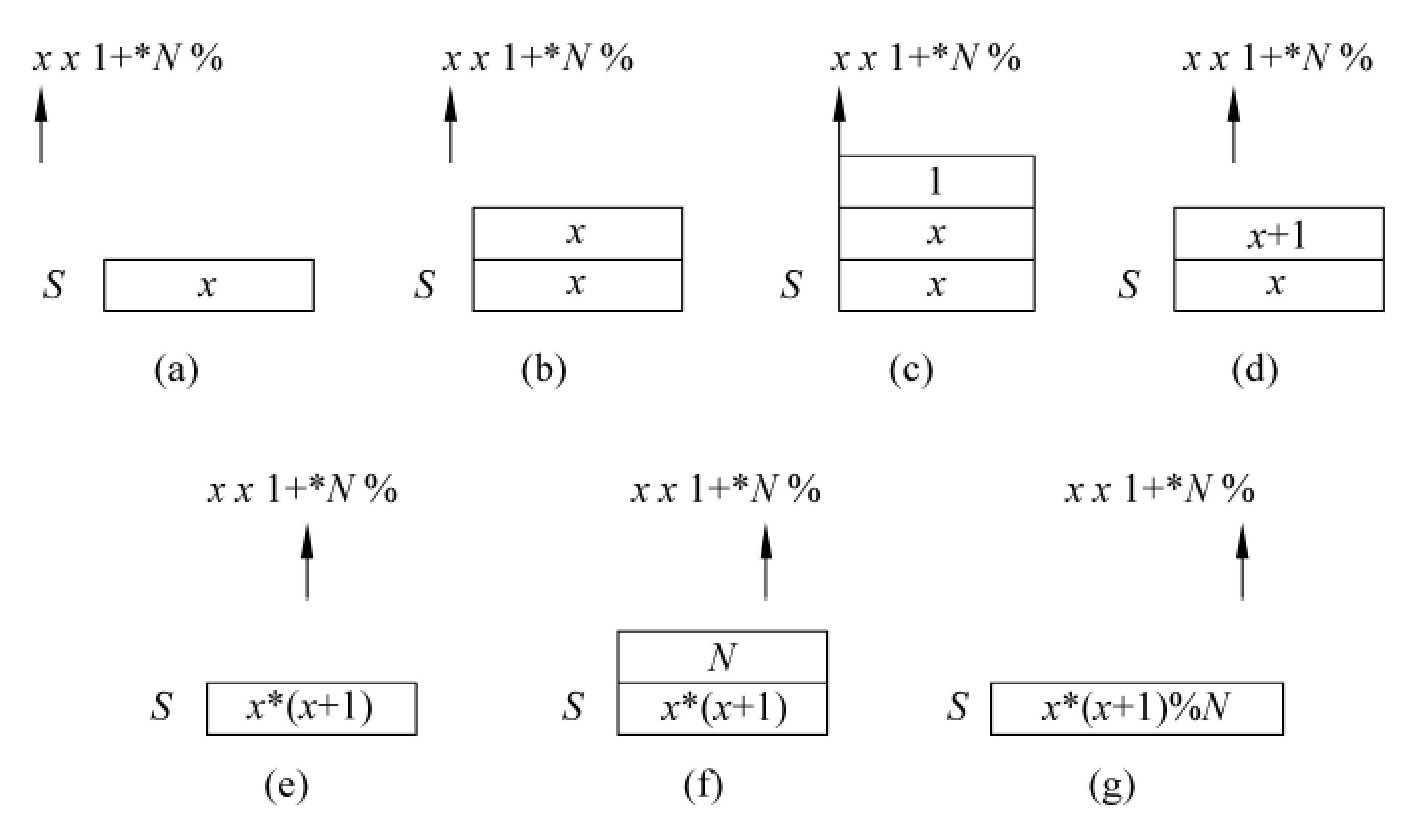


图 2-8　利用栈计算逆波兰式

把根据逆波兰式 RPN、自变量值 x 与模 N 计算出 $f(x)$ 的过程表示为下列的伪代码。

```
CALCULATE(RPN,x,N)
S←∅                                      ▷设置空栈
while RPN≠∅
  do d←read a item from RPN
     if d∈{'+','*','%'}                  ▷若 d 是运算符
       then op2←POP(S)                   ▷从栈 S 中弹出第 2 个运算数
         op1←POP(S)                      ▷从栈 S 中弹出第 1 个运算数
         if d='+'                        ▷若运算符为+
           then PUSH(S,op1+op2)          ▷计算两数之和并压入栈中
           else if d='*'                 ▷若运算符为*
                  then PUSH(S,op1*op2)   ▷将两数之积压栈
                  else PUSH(S,op1%op2)   ▷运算符为%
       else PUSH(S,d)                    ▷d 是运算数
return POP(S)
```

算法 2-8　解决 Eventually periodic sequence 问题中计算逆波兰表达式的算法过程 CALCULATE

利用 CALCULATE 过程,把解决 Eventually periodic sequence 问题的算法写成如下伪代码。

```
EVENTUALLY-PERIODIC-SEQUENCE()
打开输入文件 inputdata.txt,其为 f1
打开输出文件 outputdata.txt,其为 f2
while true
  do s←read a line from f1
     N←read the first item of s
     x←read the seconed item from s
     if x=0 and N=0
       then return
     i←0,RPN←s 中剩余部分
     f[i]←x←CALCULATE(RPN,x,N)
```

```
    while f[i]∉{f[0],…,f[i-1]}
      do i←i+1
      f[i]←x←CALCULATE(RPN,x,N)
    write i-k into f2 as a line,where k∈{1..i-1} that f[k]=f[i]
```

算法 2-9　解决 Eventually periodic sequence 问题的算法过程 EVENTUALLY-PERIODIC-SEQUENCE

2. 程序实现

利用程序 2-12 和程序 2-13，可把解决 Eventually periodic sequence 问题的算法 2-8 编写为以下程序。

```
unsigned calculate(StrInputStream *rpn,unsigned long long x,unsigned long long N){
    Stack * S=createStack(sizeof(unsigned long long));          /* 创建空栈 */
    char d[20];                                                 /* 运算对象子串 */
    unsigned long long op1,op2,r;                               /* 运算数和运算结果 */
    while(!sisEof(rpn)){                                        /* rpn 非空 */
        readString(rpn,d);                                      /* 分析当前运算对象 */
        if(strcmp(d,"+")==0 || strcmp(d,"*")==0 || strcmp(d,"%")==0){ /* 运算符 */
            op2=*((unsigned long long *)(pop(S)->key));         /* 弹出运算数 */
            op1=*(unsigned long long *)(pop(S)->key);           /* 弹出运算数 */
            if(*d=='+')                                         /* 是"+" */
                r=op1+op2;                                      /* 计算和 */
            else if(*d=='*')                                    /* 是"*" */
                r=op1 * op2;                                    /* 计算积 */
            else
                r=op1%op2;                                      /* 是"%",求余数 */
            push(S,&r);                                         /* 运算结果压栈 */
        }else{                                                  /* 是运算数 */
          if(strcmp(d,"x")==0)
            push(S,&x);
          else if(strcmp(d,"N")==0)
            push(S,&N);
          else{
            r=atoi(d);                                          /* 转换为整数 */
            push(S,&r);                                         /* 运算数压栈 */
          }
        }
    }
    r=*(unsigned long long *)(pop(S)->key);                     /* 栈中唯一元素为函数值 */
    clrStack(S,NULL);                                           /* 清理栈空间 */
    free(S);
    return r;                                                   /* 返回函数值 */
}
```

程序 2-14　实现算法 2-8 的 C 函数

对程序 2-14 的说明如下。

(1) 函数 calculate 实现算法 2-8 中的 CALCULATE 过程。与算法过程一样有 3 个参数:表示 $f(x)$的逆波兰表达式 rpn,初始时值为输入 n 的自变量 x,以及作为模运算的模 N。rpn 是作为一个串表示表达式的,其中又含有若干个需要读取的数据项,且类型各不相同,所以把参数 rpn 定义成第 1 章中程序 1-4 和程序 1-5 开发的串输入流类型 StrInputStream。虽然从题面上知 x 和 N 的类型应为 **unsigned**,但是在计算 $f(x)$的过程中 x 和 N 都需要压入一个工作栈中,而压入栈中的数据还包括 rpn 中的各运算数参加运算的中间结果,这些数据可能超出 **unsigned** 的取值范围,所以栈中元素的存储宽度应为 **unsigned long long** 宽度。又由于通用栈压栈操作中是用内存复制的方法替代赋值,所以需要把作为参数的 x 和 N 也定义成 **unsigned long long** 类型。函数返回自变量为 x 时 $f(x)$的值,由于表达式的最终值是模为 N(N≤11000000)模运算结果,所以返回值类型设为 **unsigned**。

(2) 与算法过程一样,函数内部需要设置用来从 rpn 中读取数据项的变量 d,在计算过程中使用的栈 S 和表示操作数的变量 op1 和 op2。由于开发的通用栈只能将指定地址内数据压栈,所以需要设置一个变量 r 用来存储计算过程中的结果。由于逆波兰表达式中既包含运算数又包含运算符,所以每次先从中读取的数据项 d 的类型定义为一个能表示字符串字符数组。对于运算数,由于两个非负整型数(32b)的和或积可能发生上溢,所以运算数 op1、op2 和运算结果 r 定义成非负的 **long long** 型(64b)。

(3) 第 2 行调用函数 createStack 创建一个在程序 2-12 定义的栈结构对象 S,压入栈的数据根据前面的讨论可知应为 **unsigned long long** 类型。然后根据 d 所含的内容来判定读到的运算对象是运算数还是运算符。

第 5～27 行的 **while** 循环实现算法过程 CALCULATE 中的第 2～12 行的 **while** 循环。其中,第 7 行用表达式 strcmp(d,"+")==0 ‖ strcmp(d,"*")==0 ‖ strcmp(d,"%")==0 表示 d ∈{'+','*','%'}。程序中第 8～17 行对应于算法过程中的第 5～11 行的处理运算符的代码十分接近,而对应于算法过程第 12 行将运算数压栈的操作,程序中却表示成第 18～26 行的嵌套分支语句。原因是 rpn 中的运算数包含符号运算数 x 和 N 以及字面运算数,需要区别对待。

接下来实现算法 2-9 的 EVENTUALLY-PERIODIC-SEQUENCE 过程。

```
int main(){
  unsigned x,N;
  char s[256];                                  /*存储表达式的串*/
  FILE *in=fopen("chap02/Eventuallyperiodicsequence/inputdata.txt","r"),
       *out=fopen("chap02/Eventuallyperiodicsequence/outputdata.txt","w");
  assert(in&&out);
  fgets(s,255,in);                              /*从输入文件中读取一行数据*/
  while(1){
      StrInputStream rpn;
        unsigned *f;
        int i=0,k=-1;
        initStrInputStream(&rpn,s);             /*用 s 初始化串输入流 rpn*/
        readInt(&rpn,&N);                       /*提取 N*/
```

```
14          readInt(&rpn,&x);                         /* 提取 x */
15          rpn.begin=rpn.current;
16          if(x==0&&N==0)                            /* N、x 均为 0,运行结束 */
17              break;
18          assert(f=(unsigned *)malloc((N+1)*sizeof(unsigned)));  /* 为数组 f 分配空间 */
19          f[i]=x=calculate(&rpn,x,N);               /* 计算 f(x) */
20          while(k<0){                               /* f[0..i]中无重复元素 */
21              sisRewind(&rpn);                      /* 串输入流复原 */
22              f[++i]=x=calculate(&rpn,x,N);         /* 计算 fi+1(x) */
23              k=linearSearch(f,sizeof(unsigned),i,&x,unsignedGreater);
24          }
25          free(f);
26          fprintf(out,"%d\n",i-k);                  /* 记录周期 */
27          fgets(s,255,in);                          /* 从输入文件中读取一行数据 */
28      }
29      fclose(in);fclose(out);
30      return EXIT_SUCCESS;
31}
```

程序 2-15　实现解决 Eventually periodic sequence 问题的 C 程序源代码

对程序 2-15 的说明如下。

(1) 将算法 EVENTUALLY-PERIODIC-SEQUENCE 过程实现为 main 函数。与算法过程一样,函数没有参数。

(2) 与算法一样,函数中设置变量 x 和 N 表示 f(x)的自变量和用来做模运算的模。文件指针 f1 和 f2 分别指向输入文件和输出文件。

(3) 函数代码的结构与算法代码的结构十分接近。第 12 行将从输入文件中读到的文本行 s 初始化为一个输入流 rpn,便于从中读取数据。

第 13～14 行读取头两个数据 N 和 n 后,第 15 行将输入流的起点定位于其后的 RPN 的起始位置。

第 18 行为存储序列 $f(x),f^2(x),\cdots,f^k(x),\cdots$ 分配数组 f 空间。由于函数是 $\{0,1,\cdots,N\}$ 到自身的映射,所以至多有 $N+1$ 个连续各不相同的值。

第 19 行调用函数 calculate 根据函数的逆波兰表达式 rpn 计算其在 x 值处的函数值 $f(x)$,并将计算结果分别赋予 x 和 f[0](注意 i 在第 11 行被初始化为 0)。这样,在第 20～24 行的 **while** 循环的循环体内第 22 行调用函数 calculate 计算函数在 x 处的值就相当于计算 $f^2(x),\cdots,f^i(x),\cdots$ 并存放于数组 f 的对应元素。对计算出来的 $f^i(x)$ 的值,第 23 行调用第 1 章的程序 1-3 中的 linearSearch 函数检测 $f^i(x)$ 是否在 $f(x),f^2(x),\cdots,f^{i-1}(x)$ 出现过。该函数在数组 f[0..i−1]中查找值为 x($x=f^i(x)$)的元素,若找到,返回该元素的下标,否则返回−1。将返回值赋予 k(在第 11 行初始化为−1),本循环的循环条件为 k<0,也就是 $f^i(x)$ 是未在 $f(x),f^2(x),\cdots,f^{i-1}(x)$ 出现过。

程序 2-14 和程序 2-15 写在文件夹 chap02\Eventually periodic sequence 中的源代码文件 Eventuallyperiodicsequence.c 中。

2.3　队列

2.3.1　队列的概念及其链表实现

队列的 FIFO 性质使得对它的操作就像注册办公室前人们排成的一列队伍。队列有一个**队首**和一个**队尾**。当一个元素入队时,它位于队列之尾,就像新到来的学生排在队尾。出队的元素总是在队首,就像等待最久的排在队首的学生。把对队列的 INSERT 操作称为 ENQUEUE,而把 DELETE 操作称为 DEQUEUE。如同栈操作 POP,DEQUEUE 不带有元素参数。

图 2-9 说明了用一个链表 L 实现的队列 Q。一个队列具有属性 $head[Q]$,它指向其队首。属性 $tail[Q]$则指向队尾。队列中的元素位于 $head[Q],\cdots,tail[Q]$。当 $head[Q]=tail[Q]=NIL$ 时,队列为空。当队列为空时,出队的试图导致队列的下溢。用链表实现队列不会产生上溢。

图 2-9(a)是具有 5 个元素的队列。图 2-9(b)是调用了 ENQUEUE(Q,17)、ENQUEUE(Q,3)和 ENQUEUE(Q,5)后该队列的格局。图 2-9(c)是调用 DEQUEUE(Q)后该队列的格局。DEQUEUE(Q)返回作为队首的关键值 15。新的队首具有关键值 6。

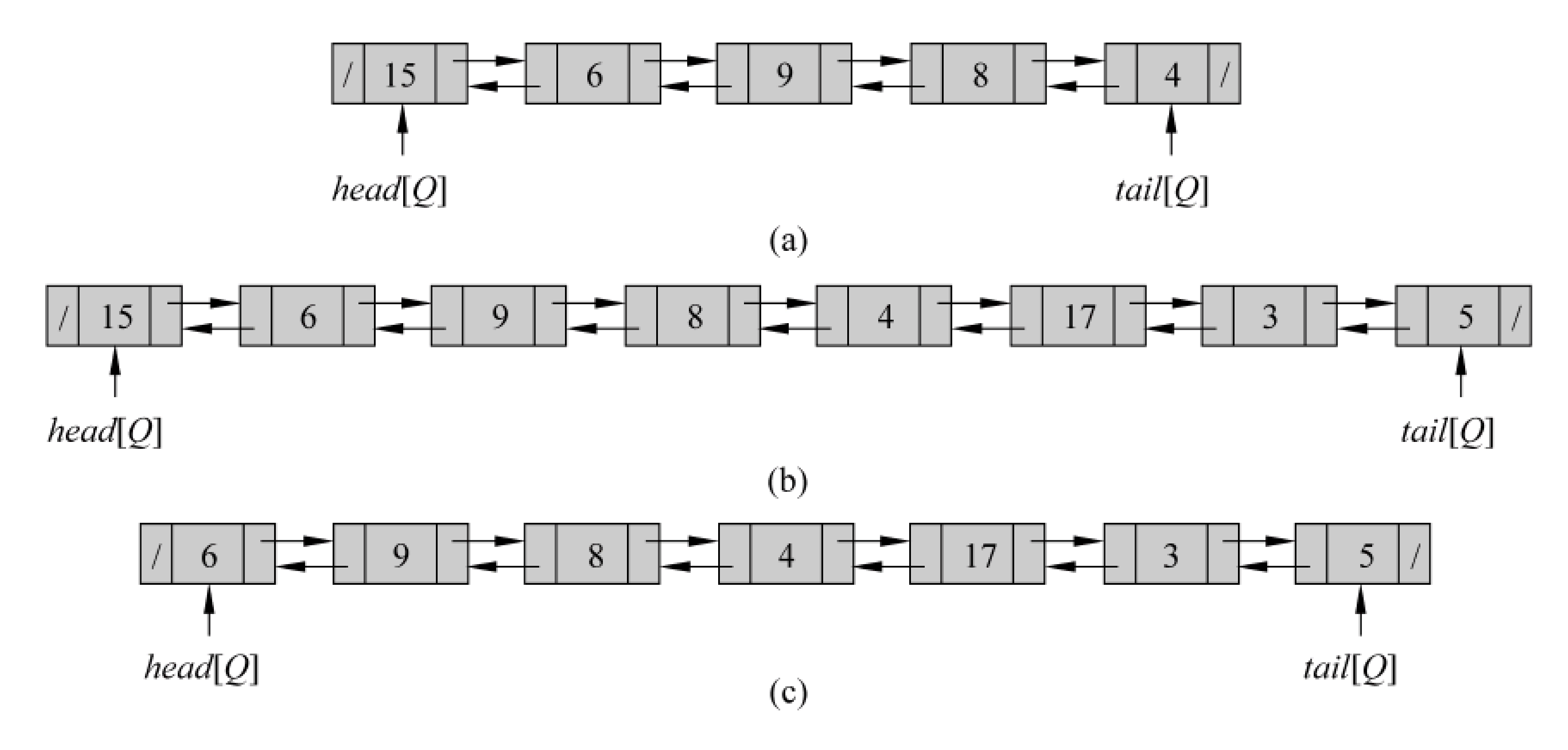


图 2-9　一个用链表 Q 实现的队列

利用对带有哨兵结点的链表的操作,队列的操作非常简单。

```
ENQUEUE(Q,x)
LIST-INSERT(Q,nil[L[Q]],x)
if head[Q]= nil[L[Q]]                ▷原队列为空
   then head[Q]←next[nil[L[Q]]]
tail[Q]←prev[nil[L[Q]]]
DEQUEUE(Q)
if Q =∅
   then error "underflow"
```

```
x←head[Q]
head[Q]←next[x]
LIST-DELETE(Q,x)
return x
```

算法 2-10　队列的入队和出队操作算法过程 ENQUEUE 和 DEQUEUE

图 2-9 说明了 ENQUEUE 和 DEQUEUE 的效果。每一个操作耗时都是 $O(1)$。

2.3.2　队列的程序实现

在 C 语言中，定义的基于链表 LinkedList 的队列数据结构如下。

```
1 #include "list.h"
2 typedef struct{                                        /*队列类型*/
3     LinkedList *L;                                     /*链表属性*/
4     ListNode *head, *tail;                             /*队首、队尾属性*/
5 }Queue;
6 Queue* createQueue();                                  /*创建空队列*/
7 void clrQueue(Queue *Q,void(*proc)(void*));            /*清理队列空间*/
8 int queueEmpty(Queue *Q);                              /*检测队列空*/
9 void enQueue(Queue *Q, void *k,int size);              /*入队操作*/
10 ListNode *deQueue(Queue *Q);                          /*出队操作*/
```

程序 2-16　定义基于链表的队列类型及声明队列操作函数的头文件 queue.h

对程序 2-16 的说明如下。

(1) 第 2～5 行定义了队列类型 Queue。它含有一个作为元素存储载体的链表类型属性 L，还含有两个分别指向队首元素与队尾元素的指针属性 head 与 tail。

(2) 第 6～8 行声明的函数 createQueue、clrQueue、queueEmpty 分别用来创建一个空队列，清理不用的队列的存储空间和检测队列是否为空。

(3) 第 9 行声明的函数 enQueue 和第 10 行声明的函数 deQueue 是实现算法 2-10 的入队操作 ENQUEUE 过程和出队操作 DEQUEUE 过程。

实现上述函数声明的定义如下。

```
#include<assert.h>
#include "list.h"
#include "queue.h"
Queue* createQueue(unsigned long size){
    Queue *Q=(Queue*)malloc(sizeof(Queue));
    assert(Q);
    Q->L=createList(size,NULL);                        /*创建链表属性*/
    Q->head=Q->L->nil->next;                           /*初始化队首*/
    Q->tail=Q->L->nil->prev;                           /*初始化队尾*/
    return Q;
}
```

```
12 void clrQueue(Queue *Q,void(*proc)(void*)){          /*清理队列空间*/
13     Q->head=Q->tail=NULL;                             /*队首、队尾指针置空*/
14     clrList(Q->L,proc);                               /*清理链表空间*/
15     free(Q->L);
16 }
17 int queueEmpty(Queue *Q){                             /*检测队列空*/
18     return listEmpty(Q->L);
19 }
20 void enQueue(Queue *Q,void *k){                       /*入队操作*/
21     listPushBack(Q->L,k);                             /*在链表中为插入*/
22     if(Q->head==Q->L->nil)                            /*插入新结点前队列为空*/
23         Q->head=Q->L->nil->next;
24     Q->tail=Q->L->nil->prev;
25 }
26 ListNode *deQueue(Queue *Q){                          /*出队操作*/
27     assert(!queueEmpty(Q));                           /*防止下溢*/
28     ListNode *x=Q->head;                              /*x置为队首*/
29     Q->head=x->next;                                  /*设置新的队首*/
30     listDelete(Q->L,x);                               /*将原队首从队列中摘下*/
31     return x;
32 }
```

程序 2-17　定义队列操作的源代码文件 queue.cpp

对程序 2-17 的说明如下。

(1) 第 4～11 行定义的函数 createQueue 通过调用 createList(size,NULL)函数创建一个可加载存储宽度为参数 size 的元素的链表属性 L,由于队列中的元素无须比较,所以第 2 个参数传递 NULL。并将 head 与 tail 属性分别初始化为 L 的表首与表尾创建一个空队列。第 12～16 行定义的函数 clrQueue 对不再使用的队列 Q 调用函数 clrList 清理队列的属性 L 的空间,并释放。

(2) 第 17～19 行定义的函数 queueEmpty 通过调用 listEmpty 检测 Q 的 L 属性是否为空来判断队列 Q 是否为空。

(3) 第 20～25 行定义的函数 enQueue 实现入队过程 ENQUEUE,第 26～32 行定义的函数 deQueue 实现出队过程 DEQUEUE。实现代码与伪代码十分相似,读者可对照研读。

2.3.3　队列的应用

用队列来解决下列应用问题。

像 素 转 换

问题描述

假设以二维数组 $G[1..m,1..n]$表示一幅图像各像素的颜色,则 $G[i,j]$表示区域中点(i,j)处的颜色,颜色值为 0 到 k 的整数。下面的算法将指定点(i_0,j_0)所在的同色邻接区域

的颜色置换为给定的颜色值。约定所有与点($i0$,$j0$)同色的上、下、左、右可连通的点组成同色邻接区域。

例如，一幅 8×9 像素的图像如图 2-10(a)所示。设用户指定点(3,5)，其颜色值为 0，此时其上方 (2,5)、下方 (4,5)、右方 (3,6)邻接点的颜色值都为 0，因此这些点属于点(3,5)所在的同色邻接区域，再从上、下、左、右 4 个方向进行扩展，可得出该同色邻接区域的其他点(见图 2-10(a)中的阴影部分)。将上述同色区域的颜色替换为颜色值 7 所得的新图像如图 2-10(b)所示。

	1	2	3	4	5	6	7	8	9
1	5	4	5	4	3	1	5	1	2
2	2	5	5	3	0	1	3	2	1
3	0	3	2	3	0	0	2	3	1
4	2	0	1	0	0	0	0	2	0
5	1	0	0	0	0	3	2	0	1
6	0	1	0	2	0	0	2	2	1
7	6	5	5	0	1	0	2	1	0
8	6	3	3	4	0	0	7	4	5

(a)

	1	2	3	4	5	6	7	8	9
1	5	4	5	4	3	1	5	1	2
2	2	5	5	3	7	1	3	2	1
3	0	3	2	3	7	7	2	3	1
4	2	7	1	7	7	7	7	2	0
5	1	7	7	7	7	3	2	0	1
6	0	1	7	2	7	7	2	2	1
7	6	5	5	0	1	7	2	1	0
8	6	3	3	4	7	7	7	4	5

(b)

图 2-10　像素转换

输入数据

输入数据组织在文件 inputdata. txt 中，该文件包含若干个案例的数据。每个案例的数据由 5 个整数 m、n、i_0、j_0、*newcolor* 组成(数据间用一个空格隔开)的一行开头，前两个整数 m 和 n 表示案例中矩阵 G 的行数和列数，接着的两个整数 i_0 和 j_0 表示图像数据列阵中指定点的行标和列标，最后一个整数 *newcolor* 表示把含有点(i_0,j_0)的邻接区域中所有像素转换成新的颜色值。随后的 m 行数据每行包含 n 个整数(数据间用一个空格隔开)，表示图像数据列阵 G。

输出数据

输出数据存放在文件 outputdata. txt 中。对应每个案例的输入数据，输出转换后的图像数据列阵 G。

输入样例

```
3 3 1 1 5
1 2 3
2 2 5
2 4 2
8 9 3 5 7
5 4 5 4 3 1 5 1 2
2 5 5 3 0 1 3 2 1
0 3 2 3 0 0 2 3 1
2 0 1 0 0 0 0 2 0
1 0 0 0 0 3 2 0 1
0 1 0 2 0 0 2 2 1
6 5 5 0 1 0 2 1 0
```

6 3 3 4 0 0 7 4 5

输出样例

1 5 3
5 5 5
5 4 2
5 4 5 4 3 1 5 1 2
2 5 5 3 7 1 3 2 1
0 3 2 3 7 7 2 3 1
2 7 1 7 7 7 7 2 0
1 7 7 7 7 3 2 0 1
0 1 7 2 7 7 2 2 1
6 5 5 0 1 7 2 1 0
6 3 3 4 7 7 7 4 5

1. 算法描述

解决一个案例的像素转换问题的基本想法是利用一个队列 Q,初始时将指定像素点 (i_0,j_0)加入队列中。然后进入一个循环,每次从队列中将队首出队,设为(x,y),然后将(x,y)的颜色值置为 *newcolor*。接着在数据列阵G 中搜索(x,y)的左、右、上、下相邻像素点是否在相邻区域中,若是则将对应的像素点加入队列循环直至队列 Q 为空为止,如图 2-11 所示。

图 2-11 为对输入样例的第一个案例进行像素转换的过程。Q 表示队列,浅色阴影表示加入队列 Q 的像素。图 2-11(a)为初始时,指定像素点(2,2)加入队列。图 2-11(b)为循环的第一次重复。将队首(2,2)出队,变换颜色,并将与其相邻的像素(2,1)和(1,2)加入队列。图 2-11(c)～图 2-11(e)为循环的各次重复。

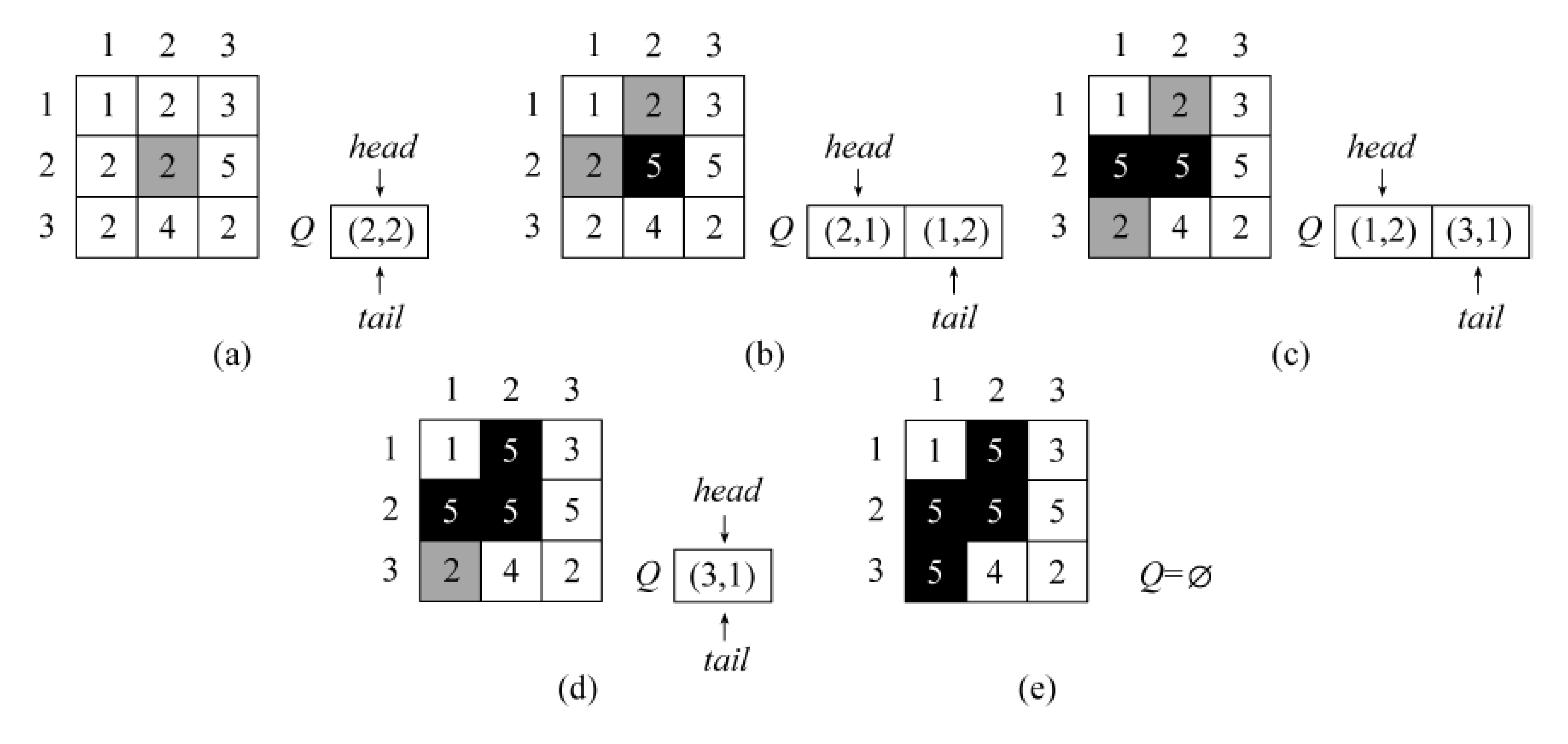


图 2-11　对输入样例的第一个案例进行像素转换的过程

将上述算法思想描述为如下的伪代码过程。

PIXEL-TRANSFORM($G,i_0,j_0,newcolor$)
1 **if** $G[i_0,j_0]=newcolor$

```
    then return
m←row[G]
n←col[G]
Q←∅                                              ▷创建空队列
oldcolor←G[i0, j0]
ENQUEUE(Q, (i0, j0))
while not QUEUE-EMPTY(Q)
  do (x, y)←DEQUEUE(Q)
     G[x, y]←newcolor
     if y-1≥1 and G[x, y-1]=oldcolor             ▷左边
     then ENQUEUE(Q, (x, y-1))
    if y+1≤n and G[x, y+1]=oldcolor              ▷右边
     then ENQUEUE(Q, (x, y+1))
    if x-1≥1 and G[x-1, y]=oldcolor              ▷上边
     then ENQUEUE(Q, (x, y-1))
    if x+1≤n and G[x+1, y]=oldcolor              ▷下边
     then ENQUEUE(Q, (x+1, y))
```

算法 2-11　解决像素转换问题的算法过程 PIXEL-TRANSFORM

2. 程序实现

利用程序 2-16 和程序 2-17，不难写出解决像素转换问题的程序。下列代码仅列出了实现算法过程 PIXEL-TRANSFORM 的函数 pixelTransform，完整的程序读者可打开文件夹 chap02\pixeltransform 的源程序文件 pixeltransform.c 研读。

```
typedef struct{                                          /*像素点的结构*/
  int x;                                                 /*行标*/
  int y;                                                 /*列标*/
}Point;
void pixelTransform(int * G,int m,int n,int i0,int j0,int newColor){
  if(G[i0 * n+j0]==newColor)                             /*新旧颜色相同*/
    return;
  Point p={i0,j0};                                       /*指定像素*/
  int oldColor=G[i0 * n+j0];                             /*指定像素的原有颜色*/
  Queue * Q=createQueue(sizeof(Point));                  /*创建空队列*/
  enQueue(Q,&p);                                         /*指定像素入队*/
  while(!queueEmpty(Q)){                                 /*只要队列非空*/
    memcpy(&p,deQueue(Q)->key,sizeof(Point));            /*队首元素赋予 p*/
    G[p.x * n+p.y]=newColor;                             /*改变颜色*/
    if(p.y-1>=0&&G[p.x * n+p.y-1]==oldColor){            /*左邻像素*/
      Point p0={p.x,p.y-1};                              /*创建像素点 p0*/
      enQueue(Q,&p0);                                    /*p0 入队*/
    }
    if(p.y+1<n&&G[p.x * n+p.y+1]==oldColor){             /*右邻像素*/
      Point p0={p.x,p.y+1};
```

```
21        enQueue(Q,&p0);
22      }
23      if(p.x-1>=0&&G[(p.x-1)*n+p.y]==oldColor){          /*上邻像素*/
24        Point p0={p.x-1,p.y};
25        enQueue(Q,&p0);
26      }
27      if(p.x+1<m&&G[(p.x+1)*n+p.y]==oldColor){           /*下邻像素*/
28        Point p0={p.x+1,p.y};
29        enQueue(Q,&p0);
30      }
31    }
32 }
```

程序 2-18　解决像素转换问题的 C 程序源代码文件 pixeltransform.c

对程序 2-18 的说明如下。

(1) 函数 pixelTransform 与算法过程 PIXEL-TRANSFORM 相比,表示图像数据列阵二维数组 G 被声明成了一维数组。这是因为对一维数组元素的访问效率高于对二维数组元素的访问。我们约定:按行优先原则,用一维数组表示二维数组。此外,多了两个表示列阵 G 的行数和列数的参数 m 和 n,这是因为 C 语言中数组不具有长度(元素个数)的属性,需要加以说明。这样,在代码中 G[i * n+j]表示二维数组元素 G[i][j]。

(2) 第 6~9 行定义了像素点数据类型,便于像素点作为队列元素的入队和出队操作。

(3) 由于 C 语言中的数组下标是从 0 开始编码的,所以列阵 G 的行标范围为 0~m-1,列表范围为 0~n-1 与算法过程中 1~m 及 1~n 稍有不同,这体现在第 20 行、第 24 行、第 28 行和第 32 行的对像素 p 的相邻像素的边界检测上。程序代码与伪代码十分相近,读者可比较研读。

2.4　二叉搜索树

2.4.1　二叉树及其在计算机中的表示

1. 二叉树概念

二叉树是递归定义的。一棵**二叉树** *T* 是一个定义在有限结点集合上的结构,它不包含结点或由 3 个不相交的结点集合组成:一个**根**结点、一棵称为**左子树**的二叉树和一棵称为**右子树**的二叉树。

不含结点的二叉树称为**空树**或**零树**,有时表示为 NIL。若左子树非空,其根称为整棵树的根的**左孩子**。类似地,非空右子树的根是整棵树的根的**右孩子**。若一棵子树是零树 NIL,我们称那个孩子缺席或缺少。图 2-12(a)展示了一棵二叉树。

在一棵二叉树中,若一结点仅有一个孩子,要讲究该孩子的位置是**左孩子**还是**右孩子**。图 2-12(b)展示了一棵由于结点位置而与图 2-12(a)中的树不同的二叉树。

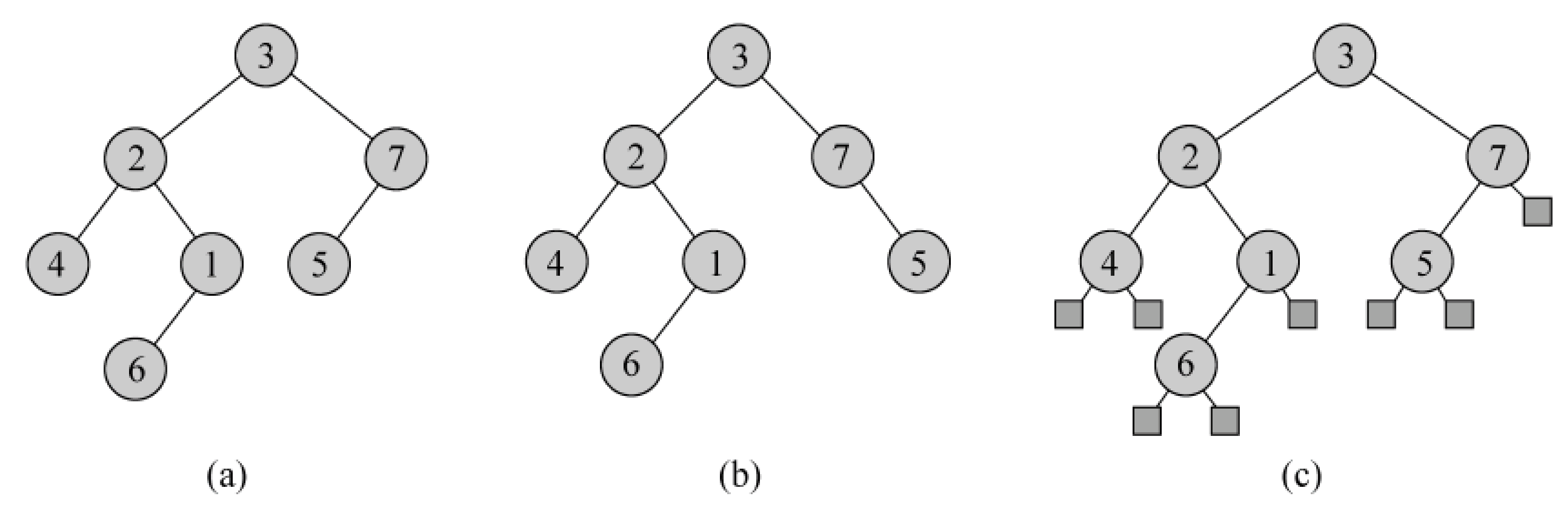


图 2-12　二叉树

图 2-12(a)为一棵标准形式的二叉树。结点的左孩子画在结点的左下方,右孩子画在结点的右下方。图 2-12(b)为一棵与图 2-12(a)中的二叉树不同的二叉树。在图 2-12(a)中,结点 7 的左孩子是 5 而其右孩子缺失。在图 2-12(b)中,结点 7 的左孩子缺失而右孩子是 5。作为二叉树,它们是不同的。图 2-12(c)为用一棵满二叉树表示图 2-12(a)中的二叉树。满二叉树是指一棵有序树:每个内结点的度数为 2。在此树中叶子表示为方形的。

二叉树中位置的信息可以用一棵有序树的内结点来表示,如图 2-12(c)所示。其思想是将二叉树中缺失的孩子用一个没有孩子的结点替代。这些叶子结点在图中表示为方形。结果的树是一棵**满二叉树**:每一个结点或是叶子或度数为 2。因此,结点孩子的顺序保存了位置信息。

一棵完全二叉树的所有叶子具有相同的深度并且所有的内结点度数为 2。图 2-13 展示了一棵高度为 3 的完全二叉树。一棵高度为 h 的完全二叉树有多少片叶子呢?根结点有 2 个深度为 1 的孩子,其中的每一个有 2 个深度为 2 的孩子,依此类推。于是,深度为 h 的叶子的数目是 2^h。因此,具有 n 片叶子的完全二叉树的高度为 $\lg n$。高度为 h 的完全二叉树的内结点数目是

$$1+2+2^2+\cdots+2^{h-1}=\sum_{i=0}^{h-1}2^i=2^h-1$$

于是,完全二叉树有 2^h-1 个内点。

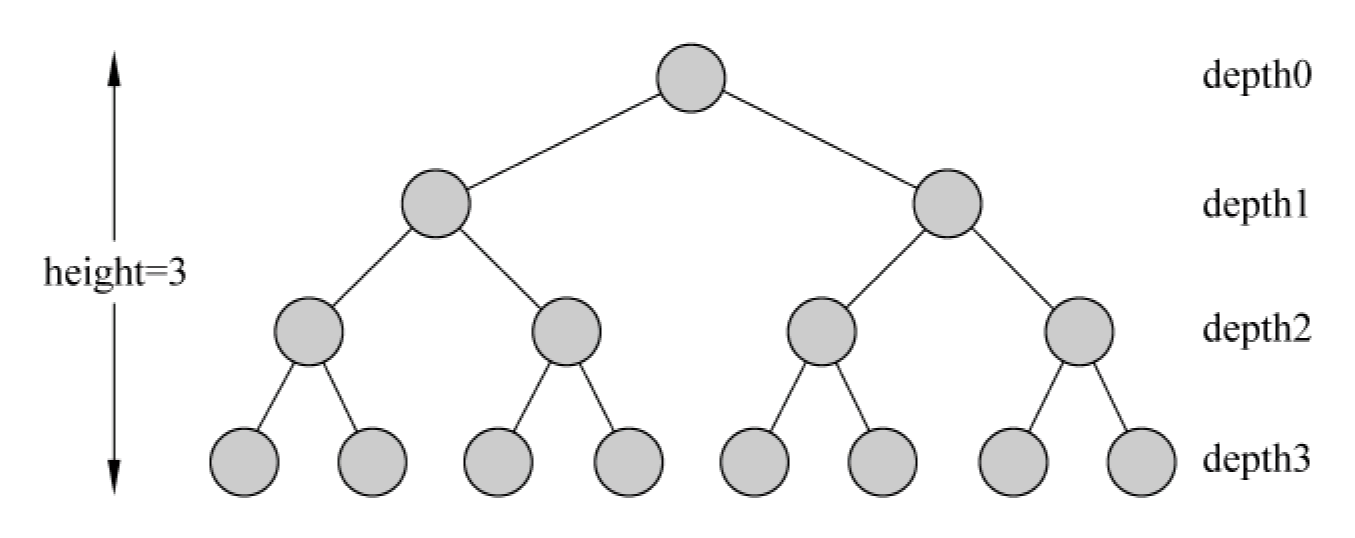


图 2-13　一棵高度为 3 的完全二叉树有 8 片树叶 7 个内结点

在计算机中,这样的一棵树可以用链接数据结构来表示。在这个数据结构中,每个结点是一个对象。除了关键值域 key 和卫星数据(结点中除了关键值以外的数据统称为结点的卫星数据)外,每个结点包含域 $left$、$right$ 和 p 分别指向对应于左孩子、右孩子和父亲的结点。若缺少孩子或父亲,则结点对应的域包含值 NIL。根结点是树中唯一父亲域为 NIL 的

结点。如图 2-14 所示,使用域 p、$left$ 和 $right$ 存储树 T 中每个结点指向其父结点、左孩子结点和右孩子结点的指针。整棵树 T 的根用属性 $root[T]$指示,若 $root[T]$=NIL,则该树为空。

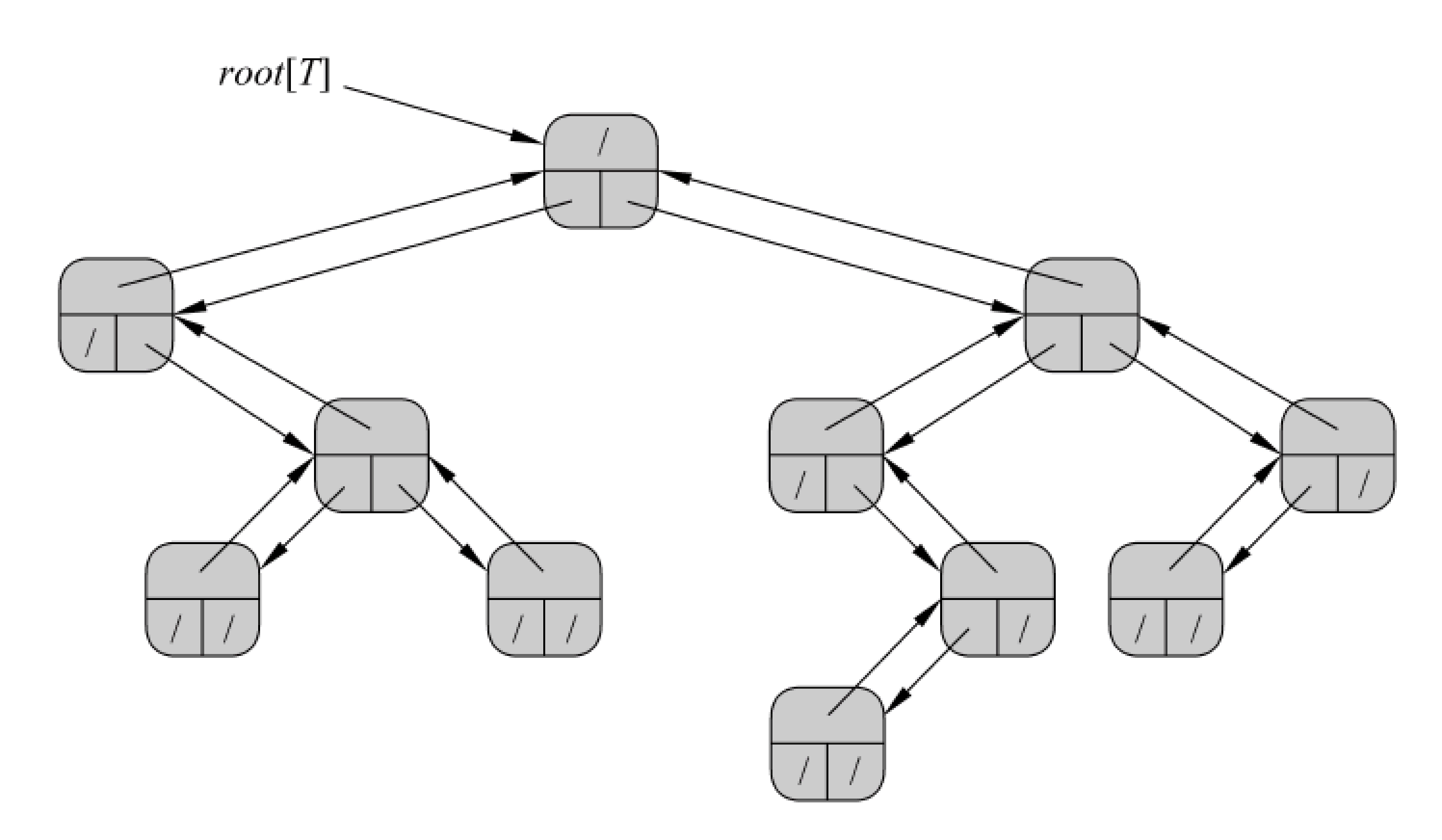


图 2-14　二叉树在计算机中的表示

图 2-14 中每一个结点 x 具有域 $p[x]$(top)、$left[x]$(左下)和 $right[x]$(右下),没有展示 key 域。

对二叉树的最基本的操作是遍历,即访问树中的每一个结点。针对二叉树的递归定义,可以按如下顺序来逐一访问树中各结点。

(1) **中序遍历**。从根结点开始,先遍历当前结点的左子树,然后访问当前结点,最后遍历右子树。

(2) **前序遍历**。从根结点开始,首先访问当前结点,接着遍历左子树,最后遍历右子树。

(3) **后序遍历** 。从根结点开始,先遍历左子树,再遍历右子树,最后访问当前结点。

下列伪代码过程表示了中序遍历,前序遍历和后序遍历过程读者可仿照描述。

```
INORDER-TREE-WALK(x)
if x≠NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

算法 2-12　对以 x 为根结点的二叉树进行中序遍历的算法过程 INORDER-TREE-WALK

遍历一棵具有 n 个结点的二叉搜索树耗时 $\Theta(n)$,这是因为初始调用后,该过程对树中的每个结点恰被调用两次——一次为其左孩子,另一次为其右孩子。

2. C 语言中二叉树的定义

在 C 语言中,把二叉树的结点定义成如下结构体。

```
#include<stdlib.h>
typedef struct node{              /*二叉树结点类型*/
    struct node *p;               /*父结点指针*/
```

```
4      struct node  * left;          /* 左孩子结点指针 */
5      struct node  * right;         /* 右孩子结点指针 */
6      void  * key;                  /* 数据域指针 */
7 }BTreeNode;
8 BTreeNode  * creatBTreeNode(void  * key,int size);                          /* 创建结点 */
9 void clrBTreeNode(BTreeNode *  r,void( * proc)(void * ));               /* 清理二叉树结点 */
10 BTreeNode *  creatBTree(void  * k,int size,BTreeNode *  l,BTreeNode *  r);  /* 创建树 */
11 void clrBTree(BTreeNode  * r,void( * proc)(void * ));                     /* 清理二叉树 */
12 void inorderTreeWalk(BTreeNode  * r,void ( * proc)(void  * )) ;            /* 中序遍历 */
13 void preorderTreeWalk(BTreeNode  * r,void ( * proc)(void  * ));           /* 前序遍历 */
14 void postorderTreeWalk(BTreeNode  * r,void ( * proc)(void  * ));          /* 后序遍历 */
```

程序 2-19　定义二叉树及声明遍历函数的头文件 binarytree. h

对程序 2-19 的说明如下。

(1) 第 2～7 行定义了表示二叉树结点的结构体类型 BTreeNode。指针属性 p、left 和 right 分别用来指向本结点的父结点、左孩子结点和右孩子结点。指针属性 key 用来指向数据域,指针类型定义为 **void** * 是出于通用性考虑。

(2) 第 8 行的函数 creatBTreeNode 用参数 key 指向的宽度为 size 的数据创建一个二叉树结点,第 9 行的函数 clrBTreeNode 用来对不再使用的结点进行清理,避免内存泄漏,其中函数指针参数 proc 用来清理结点的 key 属性空间。第 10 行的函数 creatBTree 创建一棵数据域初始化为宽度为 size、值由 k 指引的数据,左、右孩子分别为 l 和 r 的二叉树。第 11 行的函数 clrBTree 清理二叉树的存储空间,以防内存泄漏。

(3) 第 12～14 行分别是对二叉树进行中序遍历、前序遍历和后序遍历的函数 inorderTreeWalk、preorderTreeWalk 和 postorderTreeWalk 的声明。

定义上述声明的各函数如下。

```
#include "binarytree. h"
BTreeNode  * creatBTreeNode(void  * key,int size) {                /* 创建结点 */
    BTreeNode  * x=(BTreeNode * ) malloc(sizeof (BTreeNode));    /* 根结点 */
    x->key=(void * ) malloc(size);                                 /* 为数据域分配空间 */
    memcpy(x->key,key,size);                                       /* 对数据域进行数据复制 */
    x->p=x->left=x->right=NULL;                                    /* 结点指针置空 */
    return x;
}
void clrBTreeNode(BTreeNode *  r,void( * proc)(void * )) {         /* 清理二叉树结点 */
    r->p=r->left=r->right=NULL;
    if(proc)                                                       /* 对 key 域深入清理 */
        proc(r->key);
    free(r->key);
}
BTreeNode  * creatBTree(void  * k,int size,BTreeNode *  l,BTreeNode *  r) {   /* 创建树 */
    BTreeNode  * t= creatBTreeNode(k,size);                       /* 生成根结点 */
    t->left=l;                                                     /* 设置根结点的左孩子 */
    t->right=r;                                                    /* 设置根结点的右孩子 */
```

```
19    if(l)
20     l->p=t;                                          /*设置左孩子的父结点*/
21    if(r)
22     r->p=t;                                          /*设置右孩子的父结点*/
23    return t;
24 }
25 void clrBTree(BTreeNode *r,void(*proc)(void*)){
26    if (r->left)                                         /*清理左子树*/
27          clrBTree(r->left,proc);
28    if (r->right)                                        /*清理右子树*/
29          clrBTree(r->right,proc);
30    clrBTreeNode(r,proc);                                /*清理本结点*/
31    free(r);
32 }
33 void inorderTreeWalk(BTreeNode *r,void (*proc)(void *)) {   /*中序遍历*/
34   if(!r)
35     return;
36   inorderTreeWalk(r->left,proc);                        /*中序遍历左子树*/
37  proc(r->key);                                          /*中序访问当前结点*/
38  inorderTreeWalk(r->right,proc);                        /*中序遍历右子树*/
39 }
40 void preorderTreeWalk(BTreeNode *r,void (*proc)(void *)){   /*前序遍历*/
41 ⋮
42 }
43 void postorderTreeWalk(BTreeNode *r,void (*proc)(void *)){  /*后序遍历*/
44 ⋮
45 }
```

程序 2-20 C 语言二叉树的操作函数定义文件 binarytree. c

对程序 2-20 的说明如下。

(1) 第 2~8 行定义了创建二叉树结点的函数 creatBTreeNode。该函数首先为新结点指针分配空间(第 3 行),然后为结点的数据域指针 key 分配空间(第 4 行),并调用库函数 memcpy 将参数 key 指引宽度为 size 的内存数据复制到结点的属性 key 所指引的内存地址中(第 5 行)。

将新结点的父结点、左孩子、右孩子指针初始化为空(第 6 行)后返回新结点指针。第 15~24 行定义的利用已存在的两棵二叉树 l 和 r 以及指定数据 k 创建二叉树的函数 creatBTree 调用本函数创建新的二叉树的根结点。

(2) 第 9~14 行定义的函数 clrBTreeNode 负责将不用的结点空间加以清理。由于结点的数据域 key 可能含有下一层指针域,这就需要做深入的清理,利用函数指针参数 proc 来完成对下一层动态空间的清理(第 11 行和第 12 行)。第 25~32 行定义的函数 clrBTree 调用本函数依次(后序遍历顺序)清理树中每个结点的空间。

(3) 第 33~39 行定义的函数 inorderTreeWalk 实现算法 2-10 中的对二叉树的中序遍历过程 INORDER-TREE-WALK。

（4）第 40～42 行定义的函数 preorderTreeWalk 和第 43～45 行定义的函数 postorderTreeWalk 分别实现对二叉树的前序和后序遍历。为节省篇幅省略了代码细节，读者可打开本书提供相应的源代码文件研读。

3. 应用

二叉树作为一种数据结构，可以用来作为许多应用问题的数据模型。例如，可以用二叉树来表示数学表达式，每一个双目运算可以表示成以运算符为根、第一个运算数作为左孩子、第二个运算数作为右孩子的二叉树。对于单目运算，我们约定：运算符仍表示为根，左孩子置为空 NIL，运算数为右孩子。例如，表达式 $a+b-c$ 可用图 2-15 中的二叉树表示。根据表达式二叉树的结构特点，运算符均为树中的内点，而运算数必为叶子结点。

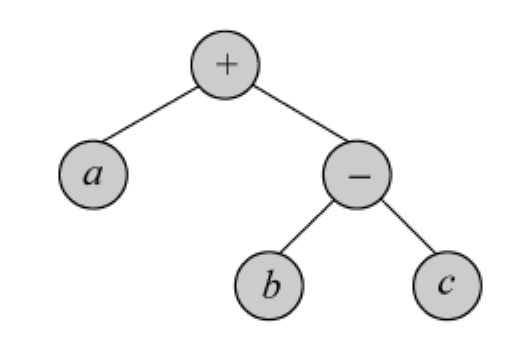


图 2-15　用二叉树来表示表达式 $a+b-c$

作为二叉树的应用，试考察下列问题。

What Fix Notation

There are three traversal methods commonly used in compilers and calculators：

prefix

infix

postfix

For example，a single expression can be written in each form

infix：　a+b * c

prefix：　+a * b c

postfix：　a b c * +

Note that prefix and postfix ARE NOT mirror images of each other! The advantage of prefix and postfix notations is that parentheses are unnecessary to prevent ambiguity.

In our traversal the following symbols are operators with precedence rules going from highest to lowest：

$	exponentiation
* /	multiply and divide
+−	add and subtract
&|	AND and OR
!	NOT

Input and Output

You are given two strings. The first string is the infix version of the expression. The second string is the prefix version of the expression. Determine the postfix version of the expression and print it out on a single line.

All input will be single characters separated by a space. Output must be the same，single characters separated by a space. There are no special sentinels identifying the end of the data.

Sample Input

```
a+b-c
+a-b c
```

Sample Output

```
INFIX    => a+b-c
PREFIX   =>+a-b c
POSTFIX  => a b c-+
```

在编译系统和计算器中需要处理表达式,表达式根据运算符与运算数的相对位置分成中缀、前缀和后缀 3 种。不同的运算符,运算的优先级是不同的。对于人们所熟悉的中缀表达式而言,需要通过加括号来改变运算顺序。而表达式一旦表示成了前缀式或后缀式,就不需要括号来辅助说明运算顺序了。我们知道,一个表达式可以构成一棵二叉树。仔细考察不难发现,对表达式二叉树进行中序、前序和后序遍历刚好生成该表达式的中缀式、前缀式和后缀式。例如,对图 2-15 所示的二叉树进行中序遍历、前序遍历和后序遍历的结果分别为

```
a+b * c
+a * b c
a b c * +
```

本问题是已知运算符的优先级以及表达式的中缀式与前缀式,要求给出表达式的后缀式。最简单的解法就是根据前缀式恢复表达式二叉树,然后对生成了的二叉树进行 3 种顺序的遍历,得到所要求的结果——表达式的 3 种形式。

设前缀式表示为串 *prefix* 中,下列算法描述了根据 *prefix* 创建表达式二叉树的过程。

```
RESTOR-TREE(prefix)
operators{'$','*','/','+','-','&','|','!'}
oprands←∅
for each item in reverse of prefix
  do left←right←NIL
     if item∈operators
       then if item ≠'!'
              then left←POP(oprands)
            right←POP(oprands)
     PUSH(oprands, EXPRESSION(item, left, right))
return TOP(oprands)
```

算法 2-13 解决 What Fix Notation 问题中根据前缀式创建二叉树的过程 RESTOR-TREE

注意,本问题中只有一个一元运算:逻辑非'!'。其他所有的运算都是二元运算。也就是说,运算符-仅表示减法,而没有负号的意义。所有的运算符存储在集合 *operators* 中,设表达式的前缀式存储于串 *prefix* 中,设置一个运算数栈 *oprands*。根据前缀式的特点:运算符表示在运算数之前,对 *prefix* 从右向左扫描,将读取到的运算数压入栈 *oprands* 中,一旦读取到运算符就从 *oprands* 中弹出运算数,合成表达式作为新的运算数压入 *oprands* 中。

第 9 行中过程 EXPRESSION(*item*, *left*, *right*)创建一棵父亲为 *item*,左、右孩子分别为 *left* 和 *right* 的二叉树。若 *prefix* 包含 n 个项,则该过程的运行时间为 $\Theta(n)$。

利用程序 2-13 和程序 2-20,在 C 语言中如下实现算法 2-13。

```
1 BTreeNode *restorTree(StrInputStream* sin){
2   char operators[]="$*/+-&|!";
3   char item[10];
4   Stack* oprands=createStack(sizeof(BTreeNode*));
5   while(!sisEof(sin)){                                     //前缀式中还有运算项
6     BTreeNode *left=NULL, *right=NULL, *opd;
7     readString(sin, item);                                 //读取一项
8     if(strstr(operators, item)){                           //是运算符
9        if(item[0]!='!')                                    //是二元运算
10          left=*(BTreeNode**)pop(oprands)->key;            //弹出左运算数
11      right=*(BTreeNode**)pop(oprands)->key;               //弹出右运算数
12    }
13    opd=creatBTree(item, 10, left, right);                 //创建二叉树
14     push(oprands,&opd);                                   //压入新的运算数
15  }
16  return *(BTreeNode**)(pop(oprands)->key);                //弹出栈中唯一的元素
17 }
```

程序 2-21　实现算法 2-13 的 C 函数

对程序 2-21 的说明如下。

(1) 第 1～17 行定义的函数 restorTree 实现算法 2-13 的 RESTOR-TREE 过程。由于需要从前缀式中逆向依次读取运算项,所以参数 prefix 定义成在第 1 章的程序 1-4～程序 1-6 中开发的串输入流 strstream 类型。该函数返回创建好的表达式二叉树(见程序 2-20)。

(2) 第 2 行将运算符集合定义成串 operators。第 3 行定义的字符数组 item 用来接收 prefix 中的数据项。第 4 行定义了栈(见程序 2-13)oprands 用来存储运算数(运算数也是表达式,所以栈中存储的是二叉树指针)。

(3) 第 5～15 行的 **while** 循环对应算法 2-13 中第 3～9 行的 **for** 循环,依次将 prefix(已经逆向)中各项读到 item 中。第 8～12 行的 **if** 语句对应算法 2-13 中第 5～8 行的 **if** 语句。注意,调用子串查找库函数 strstr 来实现检测 item∈{$, *, /, +, -, &, |, !}。该函数的原型是

```
strstr(string1, string2)
```

声明于头文件<string.h>中。其功能是在 string1 中查找 string2。若存在,返回第一个匹配位置,否则返回 NULL。

第 13 行调用程序 2-20 中的函数 creatBTree,创建新的表达式二叉树,压入栈 oprands。

(4) 为节省篇幅,此处没有列出读取文件数据调用 restorTree 函数的主函数,读者可打开文件夹 chap02\What Fix Notation 相应的源文件 WhatFixNotation.c 研读。

2.4.2　二叉搜索树

二叉搜索树是组织成一棵二叉树的全序集(集合中的任意两个元素 x 和 y,关系 $x<y$,$x=y$,$x>y$ 中有且仅有一个成立)。如图 2-16 所示,二叉搜索树的一个关键点是其存储方式满足下列**二叉搜索树性质**:

设 x 为二叉树中的一个结点。若 y 是 x 的左子树中的一个结点,则 $key[y]\leqslant key[x]$。若 y 是 x 的右子树中的一个结点,则 $key[x]\leqslant key[y]$。

于是,在图 2-16(a)中,根结点的关键值为 5,其左子树中的关键值 2、3 和 5 都不超过 5,而其右子树中的关键值 7 和 8 都不比 5 小。树中的任一结点都满足此性质。例如,图 2-16(a)中的关键值 3 不小于其左子树中的关键值 2 且不大于其右子树中的关键值 5。二叉搜索树的性质使得人们能够用中序遍历算法按排序顺序打印出树中的所有结点的关键值。作为一个例子,中序遍历按顺序 2、3、5、5、7、8 打印出图 2-16 中的两棵二叉搜索树。

在二叉搜索树中对任一结点 x,其左子树内的各关键值至多为 $key[x]$,而在其右子树中的各关键值至少为 $key[x]$。同一个集合可表示为不同的二叉搜索树。大多数搜索树操作的最坏情形运行时间正比于树的高度。图 2-16(a)为 6 个结点高度为 2 的一棵二叉搜索树。图 2-16(b)包含相同关键值,高度为 4 效率稍逊的一棵二叉搜索树。

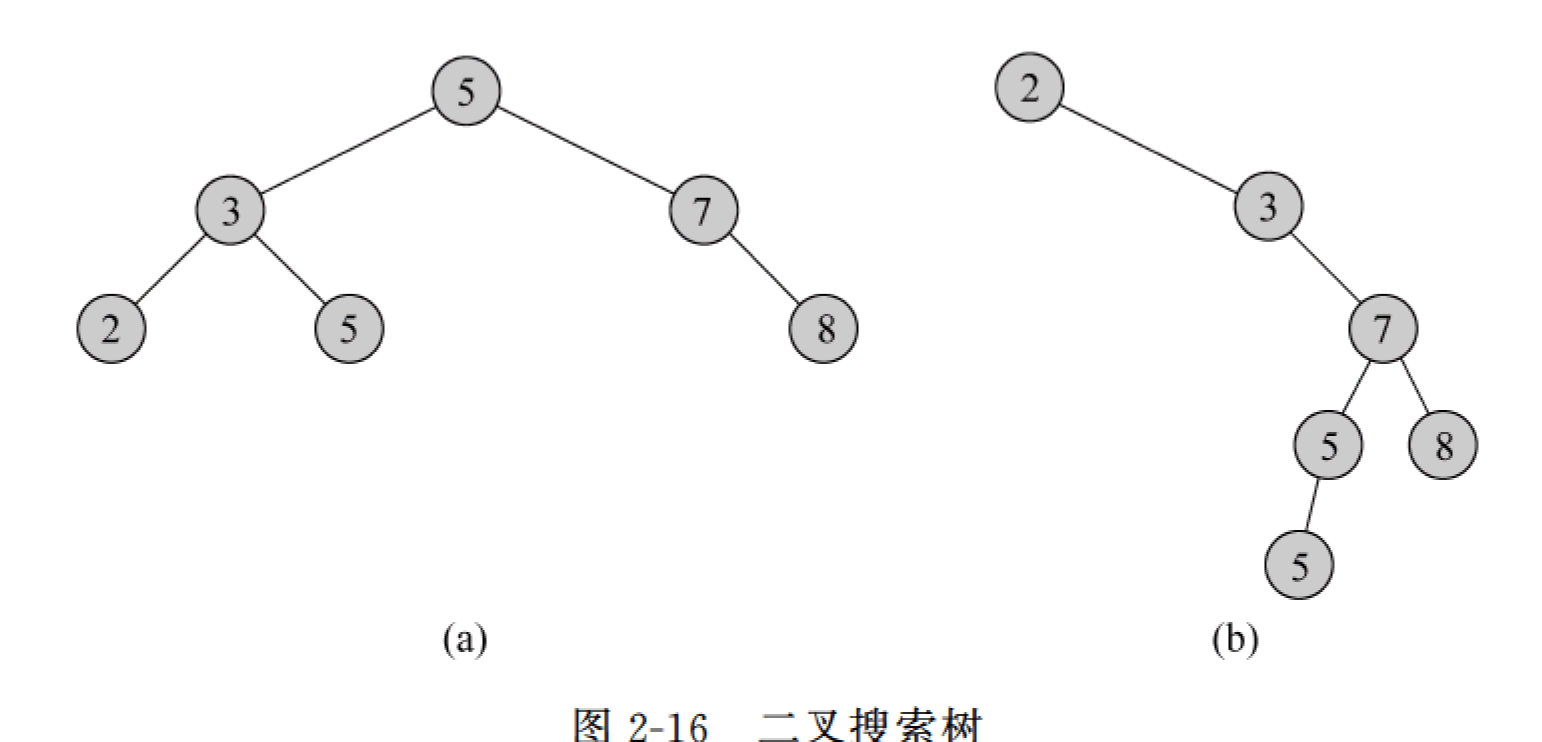


图 2-16　二叉搜索树

2.4.3　二叉搜索树的查询操作

对二叉搜索树常用的操作是对一个存储在树中的关键值的查找。除了 SEARCH 操作以外,二叉搜索树还支持 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 等查询操作。本节中,将考察这些操作并说明每个操作都能在 $O(h)$时间内被支持,其中 h 是二叉搜索树的高度。

1. 查找

在图 2-17 中,要在树中查找关键值 13,遵循从根开始的路径 15→6→7→13。树中最小关键值是 2,它可以通过从根起沿指针 *left* 找到。通过从根起沿 *right* 指针找到最大关键值 20。关键值为 15 的结点的后继是关键值为 17 的结点,这是因为它是 15 的右子树的最小关键

值。关键值为 13 的结点没有右子树，因此其后继是左孩子，也是 13 的最低的祖先。在此例中，关键值为 15 的结点就是它的后继。

人们使用下列过程在一个二叉搜索树中查找给定关键值的结点。给定一个指向该树的根的指针 x 和关键值 k，若树中存在关键值 k，返回一个指向关键值为 k 的指针；否则返回 NIL。

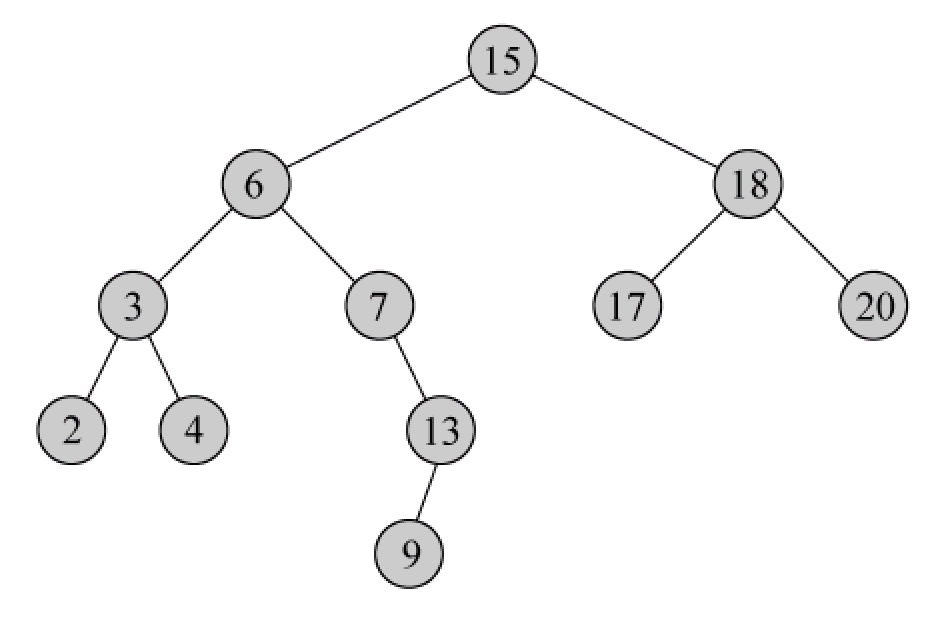


图 2-17　对二叉搜索树的查找

```
TREE-SEARCH (x,k)
while x≠NIL and k≠key[x]
  do if k<key[x]
       then x←left[x]
       else x←right[x]
return x
```

算法 2-14　在以 x 为根的二叉搜索树中查找关键值为 k 的结点的 TREE-SEARCH 过程

该过程从树根开始查找并跟踪自顶向下的一条路经，如图 2-17 所示。对所遇到的每一个结点 x，该过程将关键值 k 与 $key[x]$加以比较。若两个关键值相等，查找结束。若 k 小于 $key[x]$，继续在 x 的左子树中查找，这是因为二叉搜索树性质蕴涵着 k 不可能存储在右子树中。类似地，若 k 大于 $key[x]$，继续在右子树中查找。递归过程中遇到的结点形成树的一条从根开始向下的路径，于是 TREE-SEARCH 的运行时间是 $O(h)$，其中 h 是树的高度。

2. 最小值与最大值

二叉搜索树中关键值最小的元素总能通过从根起的跟随 $left$ 孩子直至遇到 NIL 而找到，如图 2-17 所示。下列过程返回指向以给定结点 x 为根的子树中的最小元素的指针。

```
TREE-MINIMUM (x)
while left[x]≠NIL
  do x←left[x]
return x
```

算法 2-15　计算以 x 为根的二叉搜索树中最小关键值结点的 TREE-MINIMUM 过程

二叉搜索树的性质保证 TREE-MINIMUM 是正确的。若结点 x 没有左子树，则因为 x 的右子树中的每一个关键值至少与 $key[x]$一样大，以 x 为根的子树的最小关键值为 $key[x]$。若结点 x 有左子树，则因为右子树中没有结点的关键值小于 $key[x]$且左子树中每一个关键值都不大于 $key[x]$，以 x 为根的子树中的最小值可以在以 $left[x]$为根的子树中找到。

TREE-MAXIMUM 的伪代码是对称的。

```
TREE-MAXIMUM(x)
while right[x]≠NIL
  do x←right[x]
return x
```

算法 2-16　计算以 x 为根的二叉搜索树中最大关键值结点的 TREE-MAXIMUM 过程

这两个过程对高度为 h 的树的运行时间都是 $O(h)$，如在 TREE-SEARCH 中那样，所遇到的结点序列构成从根起向下的路径。

3. 后继和前驱

给定二叉搜索树中的一个结点，能找到其在中序遍历生成的排序顺序中的后继结点有时是很重要的。若所有的关键值相互不同，结点 x 的后继是关键值大于 $key[x]$的结点中关键值最小的那一个。二叉搜索树的结构无须比较就可以确定结点的后继。当结点 x 的后继存在时，下列过程返回后继结点，若 x 的关键值是树中最大的，则返回 NIL。

```
TREE-SUCCESSOR(x)
1 if right[x]≠NIL
2    then return TREE-MINIMUM (right[x])
3 y←p[x]
4 while y≠NIL and x=right[y]
5    do x←y
6       y←p[y]
7 return y
```

算法 2-17 在二叉搜索树中计算结点 x 的(全序关系)后继的过程 TREE-SUCCESSOR

TREE-SUCCESSOR 的代码在两种情形下会中断。若结点 x 的右子树非空，则 x 的后继在右子树最左边的结点，这在第 2 行调用 TREE-MINIMUM($right[x]$)来找到。例如，图 2-17 中关键值为 15 的结点的后继是关键值为 17 的结点。

另一方面，若结点 x 的右子树为空而 x 有后继 y，则 y 是 x 的前辈中满足左孩子仍然是 x 的前辈性质的离 x 最近的那一个。在图 2-17 中，关键值为 13 的结点的后继是关键值为 15 的结点。为了找到 y，只要从 x 起向上直至遇到一个是其父亲的左孩子的结点；这由 TREE-SUCCESSOR 的第 3～7 行完成。

TREE-SUCCESSOR 对高度为 h 的树的运行时间是 $O(h)$，这是因为人们或沿树中一条向上的路径，或沿一条向下的路径。与 TREE-SUCCESSOR 对称的过程是 TREE-PREDECESSOR，其运行时间也是 $O(h)$。

即使关键值不是各不相同的，把 x 的后继与前驱分别定义为由调用 TREE-SUCCESSOR(x)和 TREE-PREDECESSOR(x)返回的结点。

2.4.4 二叉搜索树中元素的增删

插入和删除操作会导致用二叉搜索树表示的动态集合发生变化，而数据结构也会反映这一变化，但二叉搜索树的性质应仍然成立。后面将会看到，为插入一个新的元素而对树的修改是比较直接的，但对删除操作的处理稍许复杂一点。

1. 插入

为在二叉搜索树 T 中插入一个新的值 v，利用过程 TREE-INSERT。该过程传递一个结点 z，其 $key[z]=v$，$left[z]=$NIL，且 $right[z]=$NIL。它要修改 T 和 z 的一些域使得 z 插

入到树的合适的位置上。

```
TREE-INSERT(T,z)
1  y←NIL
2  x←root[T]
3  while x≠NIL
4      do y←x
5         if key[z]<key[x]
6            then x←left[x]
7            else x←right[x]
8  p[z]←y
9  if y=NIL
10    then root[T]←z              ▷树 T 是空的
11    else if key[z]<key[y]
12            then left[y]←z
13            else right[y]←z
```

算法 2-18　在二叉搜索树 T 中插入结点 z 的过程 TREE-INSERT

图 2-18 所示为在二叉搜索树中插入一个关键值为 13 的项。浅阴影结点表示从根起向下到该项要插入的位置的路径。虚线表示插入该项时要加入的连接。

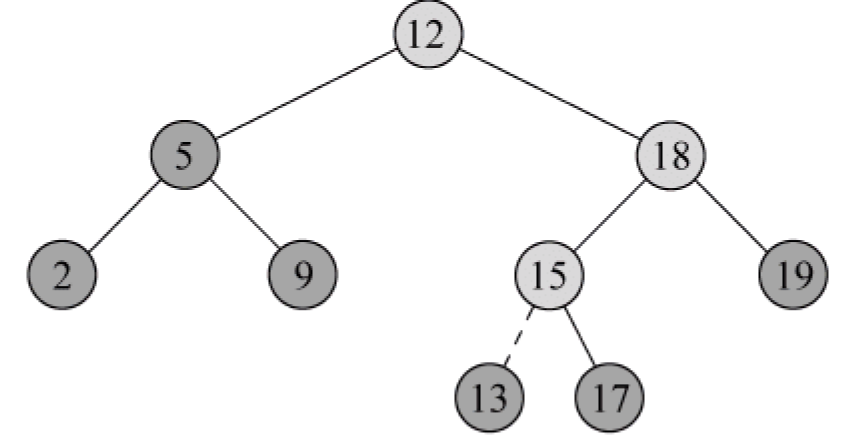


图 2-18　在二叉搜索树中插入一个关键值为 13 的项

图 2-18 展示了 TREE-INSERT 的运行。就像过程 TREE-SEARCH 那样，TREE-INSERT 从树的根开始跟踪一条向下的路径。指针 x 跟踪此路径，并维持指针 y 为 x 的父亲。初始化后，第 3～7 行的 **while** 循环导致两个指针向树的下方移动，向左还是向右取决于 $key[z]$和 $key[x]$的比较结果，直至 x 为 NIL。此 NIL 占据了希望放置输入 z 的位置。第 8～13 行设置使得 z 插入各个指针。

就像搜索树上的其他基本操作一样，过程 TREE-INSERT 在高度为 h 的树上的运行时间是 $O(h)$。

2. 删除

从二叉搜索树中删除指定结点 z 的过程以指向 z 的指针作为参数。该过程要考虑图 2-19 展示的 3 种情形。若 z 无孩子，将其父亲 $p[z]$指向 z 的指针改为 NIL。若该结点仅有一个孩子，通过在其孩子与其父亲之间建立新的连接来删除 z。最后，若该结点有两个孩子，先删掉 z 的后继结点 y，它没有左孩子(这是因为 z 有右孩子，其后继 y 为右子树中的最小者，故 y 无左孩子)，然后用 y 的关键值和卫星数据替代 z 的关键值和卫星数据。

TREE-DELETE 的代码对 3 个情形的组织稍有不同。

```
TREE-DELETE (T,z)
if left[z]=NIL or right[z]=NIL
   then y←z
   else y←TREE-SUCCESSOR(z)
```

```
4 if left[y]≠NIL
5   then x←left[y]
6   else x←right[y]
7 if x≠NIL
8   then p[x]←p[y]
9 if p[y]=NIL
10   then root[T]←x
11   else if y=left[p[y]]
12           then left[p[y]]←x
13           else right[p[y]]←x
14 if y≠z
15   then key[z]←key[y]
16        将 y 的卫星数据复制到 z 中
17 return y
```

算法 2-19 从二叉搜索树 T 中删除结点 z 的过程 TREE-DELETE

在第 1～3 行中,算法确定要删除的结点 y。结点 y 或是输入结点 z(若 z 至多有一个孩子)或是 z 的后继(若 z 有两个孩子)。然后,在第 4～6 行中,设 x 为 y 的非 NIL 孩子,若 y 没有孩子则设 x 为 NIL。在 7～13 行中通过设置 $p[y]$和 x 中的指针将 y 删除。对处理 $x=$NIL 或 y 为树根这样的边界条件,y 的删除稍微要复杂点。最后,在第 14～16 行,若删除的是 z 的后继,则要将 y 的关键值和卫星数据移到 z 处,覆盖原先的关键值和卫星数据。结点 y 在第 17 行返回,使得主调过程能将其释放到空闲表中。本过程对高度为 h 的树的运行时间为 $O(h)$。

图 2-19 为从二叉搜索树中删除结点 z。哪个是真正要删掉的结点取决于 z 有多少孩子;此结点表为浅阴影的。图 2-19(a)说明若 z 无孩子,直接删除 z。图 2-19(b)说明若 z 仅有一个孩子,删除 z。图 2-19(c)说明若 z 有两个孩子,删除其后继 y,它至多有一个孩子,然后用 y 的关键值和卫星数据替代 z 的关键值和卫星数据。

我们已经说明了对二叉搜索树的所有基本操作都运行于 $O(h)$时间,其中 h 是该树的高度。然而,随着项的插入和删除,二叉搜索树的高度是变化的。例如,若各项按严格增加的顺序一一插入,树就成了高度为 $n-1$ 的链表了。假定所有的关键值各不相同,可以将 n 个关键值随机地创建一棵二叉搜索树,其高度值可期望为 $O(\lg n)$。即使如此,对这样的一棵初始"平衡"的二叉搜索树,经过若干次插入、删除操作后很难保持它的平衡性。2.4.5 节讨论一种具有"自维护"功能的平衡二叉搜索树:在插入和删除操作后都会调整树的结构,使得既保持二叉搜索树的性质,又保持左、右子树的基本平衡。

2.4.5 红-黑树及其性质

红-黑树是一棵二叉搜索树,每个结点存储了它的**颜色**,可以是红色或者黑色。通过限制从根到叶子的路径上的结点的着色方式来保证没有任何这样的路径的长度是其他路径的两倍,使得该树大致**平衡**。

该树的每个结点包含有域 *color*、*key*、*left*、*right* 以及 p。若某结点的孩子或父亲不存

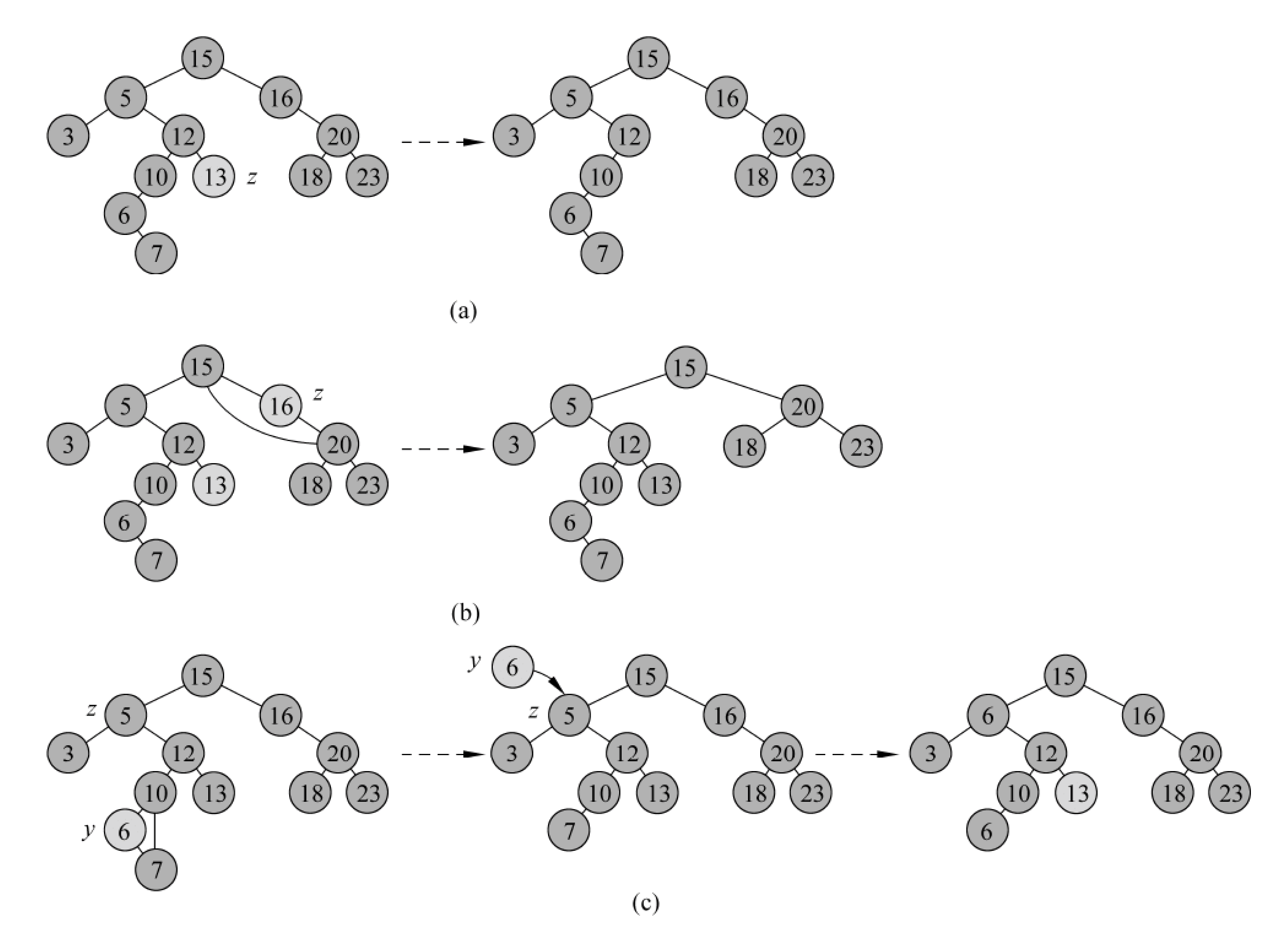


图 2-19　从二叉搜索树中删除结点 z

在，结点所含的相应的指针域包含值 NIL。我们将把这些 NIL 值视为指向二叉搜索树的外结点（树叶）的指针，而那些含有关键值的结点视为树的内结点。

一棵二叉搜索树为红-黑树，若其满足下列**红-黑树性质**。

（1）每个结点非红即黑。

（2）根是黑色的。

（3）每一片叶子是黑色的。

（4）若一结点是红色的，则其孩子是黑色的。

（5）对每个结点而言，所有从该结点起到后代叶子的路径含有同样多的黑色结点。

图 2-20 展示了一棵红-黑树的例子。

图 2-20 为一棵红-黑树，黑色结点是黑色的，红色结点带有阴影。红-黑树中的每一个结点非红即黑，红色结点的孩子都是黑色的，从一结点起到其后代叶子的每一条简单路径含有相同多的黑色结点。图 2-20(a)中每一片叶子展示为 NIL，是黑色的。每一个非 NIL 结点标注了它的黑色高度；NIL 的黑色高度为 0。图 2-20(b)中同一棵红-黑树不过把所有的 NIL 替换成单一的哨兵 $nil[T]$，它总为黑色，并忽略它的黑色高度。根的父亲也是哨兵。图 2-20(c)中同一棵红-黑树忽略所有的叶子和根的父亲。在本章其余部分将使用这一表示方式。

为方便红-黑树代码中对边界条件的处理，用单一的哨兵来表示所有的 NIL。对一棵红-黑树 T 而言，哨兵 $nil[T]$ 是一个具有树中普通结点的相同域的对象。其颜色域为

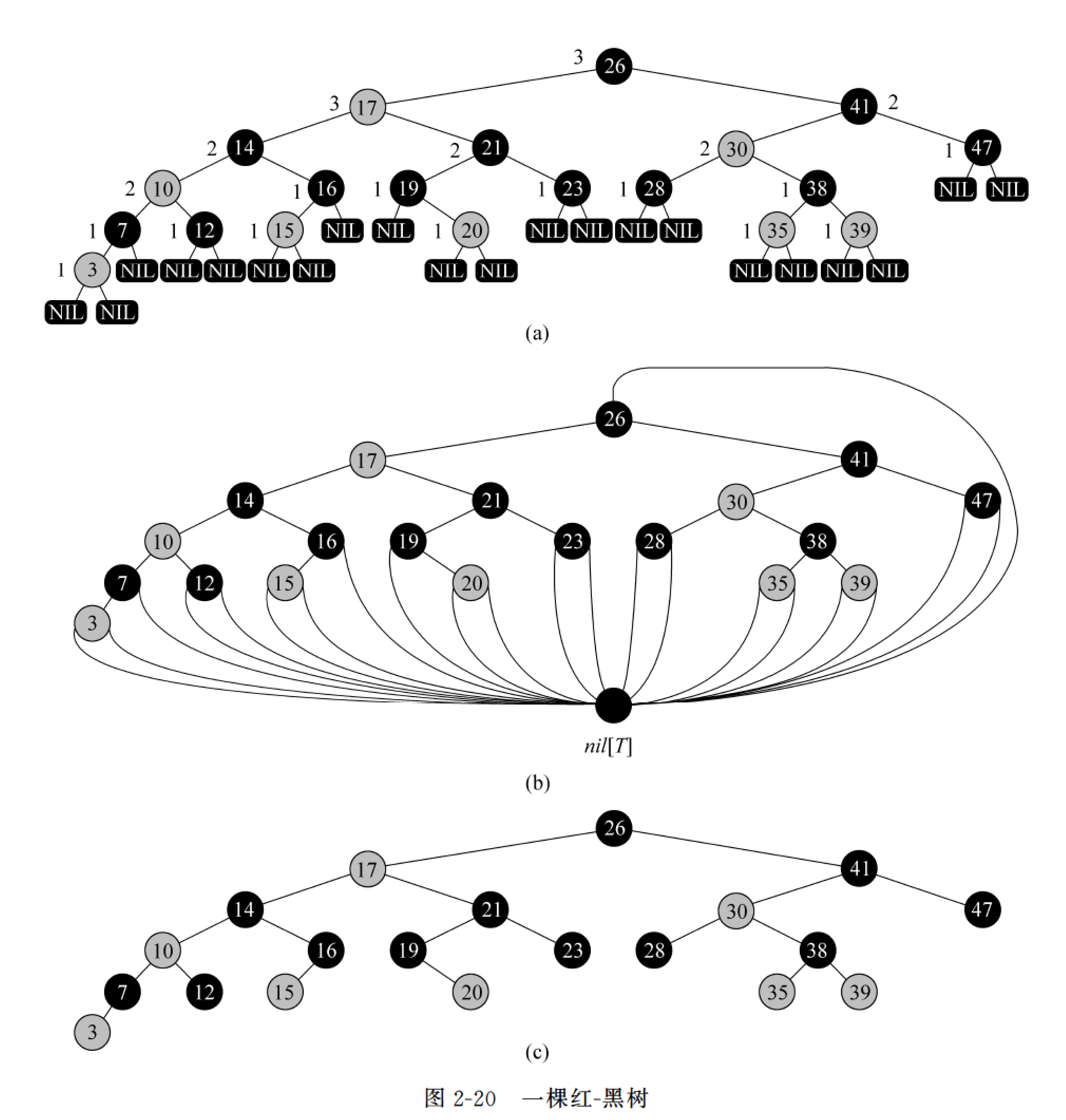


图 2-20　一棵红-黑树

BLACK,而其他域 p、$left$、$right$ 和 key 可以设为任意值。如图 2-20(b)所示,所有指向 NIL 的指针均用指向哨兵 $nil[T]$的指针代替。

利用哨兵,可以把某结点 x 的 NIL 孩子当成与普通结点一样来处理,其父亲是 x。虽然并不在树中为每个 NIL 添加一个不同的哨兵,以使得每个 NIL 的父亲都明确定义,这样的方法会浪费空间。使用一个哨兵 $nil[T]$来表示所有的 NIL——树叶和根的父亲。哨兵的域 p、$left$、$right$ 及 key 没有实质意义,但在代码过程中有时为方便会对它们加以设置。

一般来说,人们仅对红-黑树中的内结点感兴趣,因为这些结点存有关键值。在本章的余下部分,将在画红-黑树时省略叶子,如图 2-20(c)所示。

人们把从一个结点 x 起,但不包含此结点,向下到一片叶子的路径中所含的黑色结点数称为该结点的**黑色高度**,记为 $\mathrm{bh}(x)$。根据性质 5,黑色高度的概念是良好定义的,因为所有从该结点向下的路径含有相同多的黑色结点。人们定义红-黑树的黑色高度为其根的黑色高度。为说明红-黑树的高度性质,引入下列命题。

引理 2-1　红-黑树中以任何结点 x 为根的子树至少包含 $2\mathrm{bh}(x)-1$ 个内结点。

证明　对 x 的高度做数学归纳。若 x 的高度 h 为 0，则 x 必为一片叶子($nil[T]$)，以 x 为根的子树至少包含 $2^{\mathrm{bh}(x)}-1=2^0-1=0$ 个内结点。假定对于高度$<h(h>0)$的结点命题为真，考虑 x 是一个高度为 h 且具有两个孩子的内结点。每个孩子的黑色高度视其颜色是红还是黑分别为 $\mathrm{bh}(x)$或 $\mathrm{bh}(x)-1$。由于 x 的孩子的高度小于 x 本身的高度，人们可以把归纳假设施于每个孩子得出每个孩子至少有 $2^{\mathrm{bh}(x)-1}-1$ 个内结点。于是，以 x 为根的子树至少含有$(2^{\mathrm{bh}(x)-1}-1)+(2^{\mathrm{bh}(x)-1}-1)+1=2^{\mathrm{bh}(x)}-1$ 个内结点，这就证明了该命题。

引理 2-2　一棵具有 n 个内结点红-黑树的高度至多为 $2\lg(n+1)$。

证明　设 h 为树的高度，按性质 4，从根到一片叶子的任一路径上，不包括根本身，至少有一半的结点是黑色的。根据引理 2-1，根的黑色高度至少为 $h/2$；于是：

$$n \geqslant 2^{h/2}-1$$

将 1 移到左边并对两端取对数得出 $\lg(n+1)\geqslant h/2$，或 $h\leqslant 2\lg(n+1)$。

此引理的直接结果是动态集合的操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 能在 $O(\lg n)$时间内实现于红-黑树。这是因为它们可以对一棵高度为 h 的二叉搜索树运行于 $O(h)$时间内，而根据引理 2-2，任一具有 n 个结点的红-黑树高度为 $O(\lg n)$。

虽然对二叉搜索树的 TREE-INSERT 和 TREE-DELETE 对给定的作为输入的红-黑树能在 $O(\lg n)$时间内运行，但不能保证发生了改变的二叉搜索树仍将为一棵红-黑树。这就是下面要深入讨论的内容。

2.4.6　红-黑树的操作

1. 旋转

搜索树操作 TREE-INSERT 和 TREE-DELETE 对一棵红-黑树运行时，耗时 $O(\lg n)$。由于改变了树的结构，结果就可能违背红-黑树性质。为恢复这些性质，必须改变树中某些结点的颜色还要改变指针结构。

图 2-21 所示为对二叉搜索树的旋转操作。操作 LEFT-ROTATE(T,x)通过改变常数个指针将左边两个结点的格局转换为右边的格局。可以通过逆操作 RIGHT-ROTATE(T,y)将右边的格局转换成左边的格局。字母 α、β 和 γ 表示各子树。旋转保留了二叉搜索树性质：α 中的关键值先于 $key[x]$，而 $key[x]$先于 β 中的关键值，β 中的关键值先于 $key[y]$，$key[y]$又先于 γ 中的关键值。

人们通过**旋转**来改变指针结构，这是在搜索树中的保留二叉搜索树性质的局部操作。图 2-21 展示了两种情形的旋转：左旋转和右旋转。对结点 x 做一次左旋转时，假定其右孩子 y 不是 $nil[T]$；x 可以是树中任一右孩子不为 $nil[T]$的结点。左旋转将 x"轴转"连接到 y。这使得 y 为子树新的根，而 x 为 y 的左孩子而 y 的左孩子作为 x 的右孩子。

LEFT-ROTATE 的伪代码中假定 $right[x]\neq nil[T]$且根的父亲为 $nil[T]$。

```
LEFT-ROTATE(T,x)
y←right[x]                    ▷设置 y
right[x]←left[y]              ▷将 y 的左子树转换成 x 的右子树
```

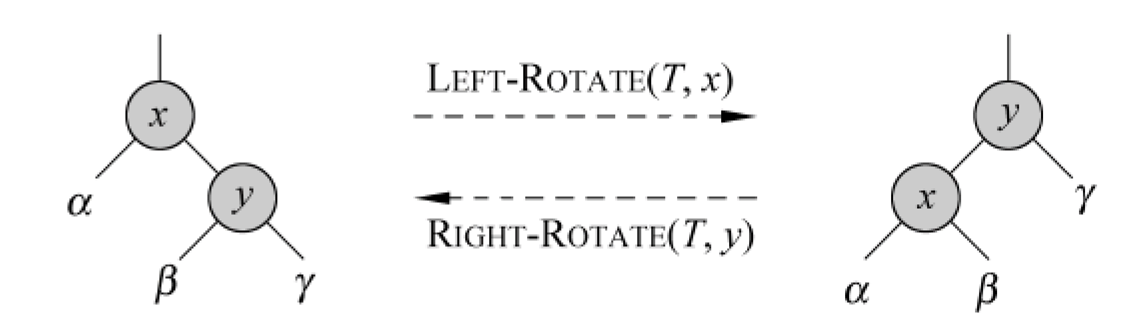


图 2-21　对二叉搜索树的旋转操作

```
p[left[y]]←x
p[y]←p[x]                          ▷将 x 的父亲连接到 y 的父亲
if p[x]=nil[T]
    then root[T]←y
    else if x=left[p[x]]
        then left[p[x]]←y
        else right[p[x]]←y
left[y]←x                          ▷将 x 置于 y 的左孩子
p[x]←y
```

算法 2-20　对红-黑树 T 中结点 x 进行左旋转操作的过程 LEFT-ROTATE

图 2-22 展示了 LEFT-ROTATE 操作,它与 RIGHT-ROTATE 的代码是对称的。LEFT-ROTATE 和 RIGHT-ROTATE 均运行于 $O(1)$时间内。旋转只改变指针,其他所有域都保持不变。

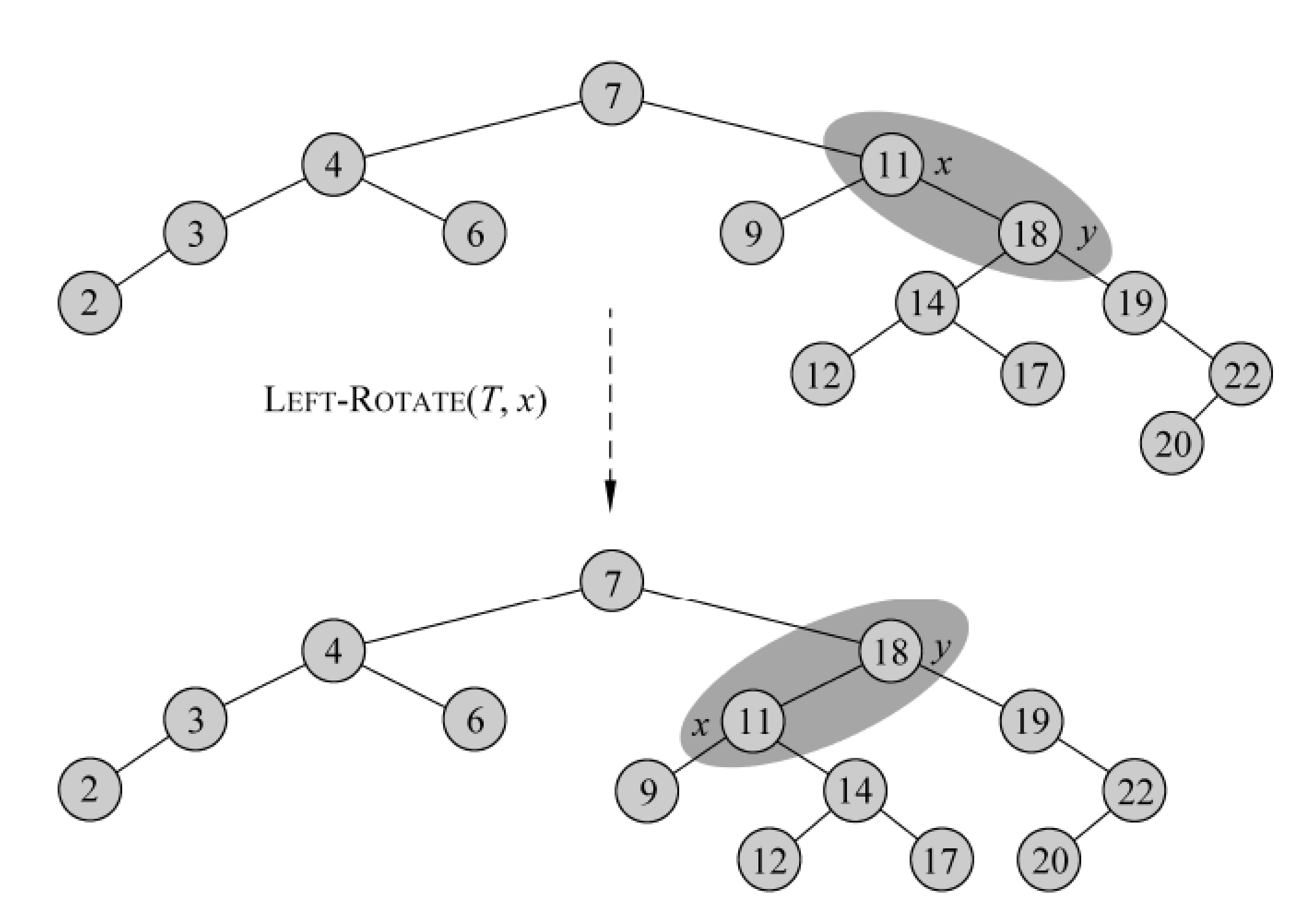


图 2-22　过程 LEFT-ROTATE(T,x)如何改变二叉搜索树的例子

2. 插入

在一棵红-黑树中插入一个结点可以在 $O(\lg n)$时间内完成。利用对 TREE-INSERT 过程的一个轻微修改版本将一个结点 z 像在普通二叉搜索中那样插入到红-黑树 T 中,然后将 z 着成红色。为保证维持红-黑树性质,调用一个辅助过程来对各结点重新着色并执行旋转。调用 RB-INSERT(T,z)将其 key 域假定已经填写好的结点 z 插入到红-黑树 T 中。

```
RB-INSERT(T,z)
1 y←nil[T]
2 x←root[T]
3 while x≠nil[T]
4     do y←x
5        if key[z]<key[x]
6           then x←left[x]
7           else x←right[x]
8 p[z]←y
9 if y=nil[T]
10    then root[T]←z
11    else if key[z]<key[y]
12            then left[y]←z
13            else right[y]←z
14 left[z]←nil[T]
15 right[z]←nil[T]
16 color[z]←RED
17 RB-INSERT-FIXUP(T,z)
```

算法 2-21　在红-黑树 T 中插入结点 z 的过程 RB-INSERT

过程 RB-INSERT 和 TREE-INSERT 之间有 4 个不同点。第一，所有的 NIL 实例都替换成了 $nil[T]$。第二，为维护正常的树结构，在 RB-INSERT 的第 14～15 行将 $left[z]$和 $right[z]$设为 $nil[T]$。第三，在第 16 行将 z 着成红色。第四，因为 z 着成红色可能导致红-黑树性质之一被违背，在 RB-INSERT 的第 17 行调用 RB-INSERT-FIXUP(T,z)来恢复红-黑树性质。

```
RB-INSERT-FIXUP(T,z)
while color[p[z]]=RED
  do if p[z]=left[p[p[z]]]
     then y←right[p[p[z]]]
        if color[y]=RED
          then color[p[z]]←BLACK            ▷情形 1
               color[y]←BLACK               ▷情形 1
               color[p[p[z]]]←RED           ▷情形 1
               z←p[p[z]]                    ▷情形 1
          else if z=right[p[z]]
               then z←p[z]                  ▷情形 2
                    LEFT-ROTATE(T,z)        ▷情形 2
                    color[p[z]]←BLACK       ▷情形 3
                    color[p[p[z]]]←RED      ▷情形 3
                    RIGHT-ROTATE(T,p[p[z]]) ▷情形 3
     else (与 then 短语一样但交换 right 和 left)
color[root[T]]←BLACK
```

算法 2-22　插入结点后恢复红-黑树性质的过程 RB-INSERT-FIXUP

为理解 RB-INSERT-FIXUP 是如何工作的，将分成 3 个步骤来考察代码。首先，要确定

RB-INSERT 将 z 插入并将其着成红色而导致的对哪一条红-黑树性质的违背。其次,将考察第 1~15 行的 **while** 循环的整体目标。最后,将探索 3 种情况[①]中的哪一个使得 **while** 循环中断,并弄清楚它们是如何完成目的的。图 2-23 展示了 RB-INSERT-FIXUP 对一个红-黑树样本的操作。

调用 RB-INSERT-FIXUP 可能对哪一条红-黑树性质有违背呢?性质 1 肯定会仍然成立,性质 3 也是如此,这是因为新插入的红色结点的孩子都是哨兵 $nil[T]$。性质 5 说的是从给定结点起的任意一条路径上含有相同多个黑色结点,也是满足的,这是因为结点 z 是红色的且带有哨兵孩子。于是,可能被违背的性质仅有性质 2,它要求根是黑色的,以及性质 4,它说的是红色结点不能有红色孩子。所有可能的违背都是因为 z 被着成红色的。若 z 是根,则性质 2 被违背,若 z 的父亲是红色的,则性质 4 被违背。图 2-23(a)展示了插入结点 z 后性质 4 被违背的情形。

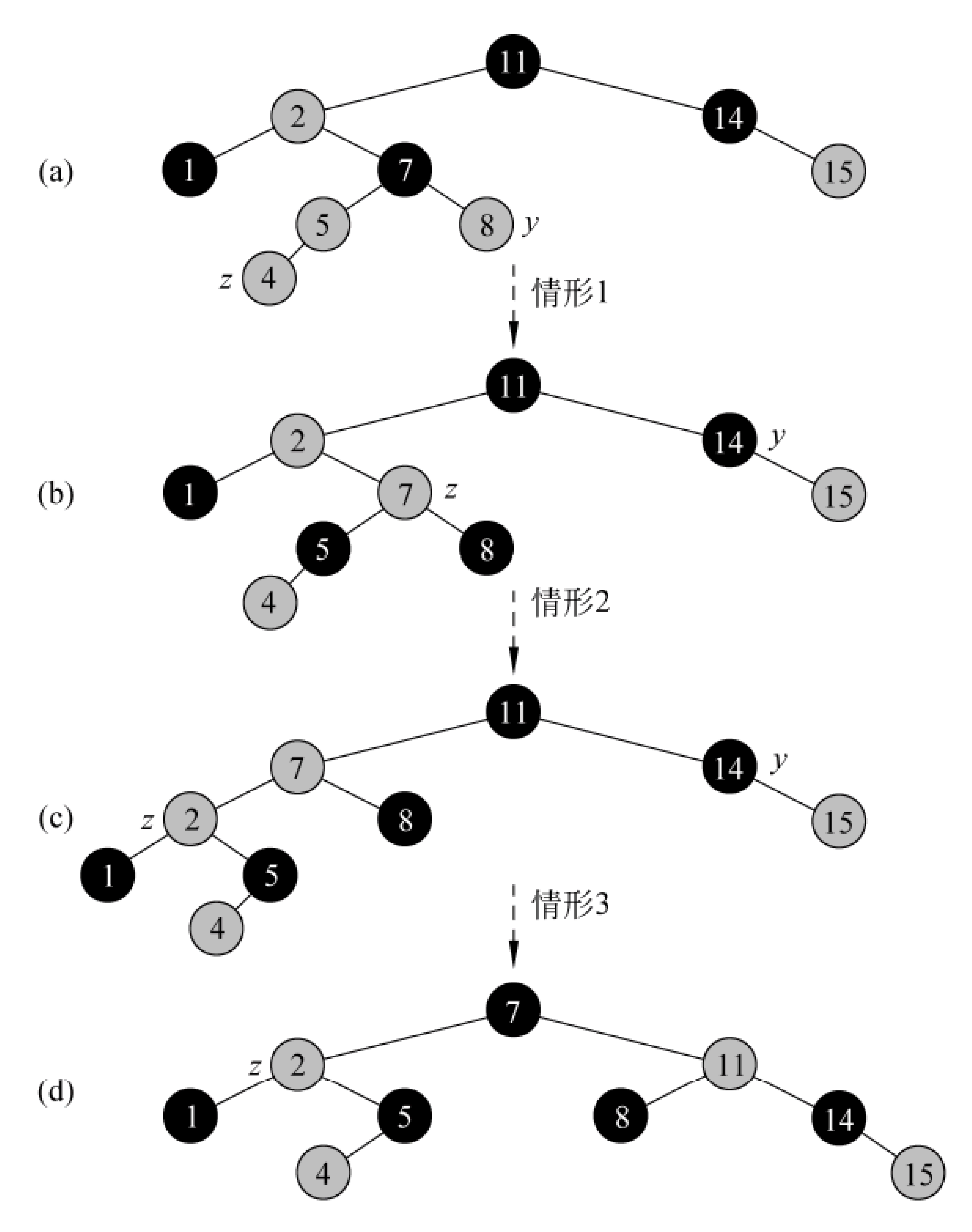


图 2-23 RB-INSERT-FIXUP 的操作

图 2-23 所示为 RB-INSERT-FIXUP 的操作。图 2-23(a)为插入后的结点 z。由于 z 及其父亲 $p[z]$都是红色的,违背了性质 4。由于 z 的叔叔 y 是红色的,应用代码中的情形 1。结点被重新着色指针 z 上移,结果显示在图 2-23(b)。同样,z 及其父亲都是红色的,但 z 的叔叔是黑色。由于 z 是 $p[z]$的右孩子,应用情形 2。执行一次左旋转,结果显示在图 2-23(c)。现在 z 是其父亲的左孩子,应用情形 3。一次又旋转得出的树在图 2-23(d)中,这是一棵合

① 情况 2 会变成情况 3,所以这两种情况并不是互斥的。

法的红-黑树。

第 1～15 行的 **while** 循环的每次重复之初，维持如下状态。

(1) 结点 z 是红色的。

(2) 若 $p[z]$是根，则 $p[z]$是黑色的。

(3) 若存在对红-黑树性质的违背，则至多有一个违背，且或是对性质 2 的违背，或是对性质 4 的违背。若违背了性质 2，那是因为 z 为根且其为红色的。若违背了性质 4，那是因为 z 和 $p[z]$都是红色的。

由于此 **while** 循环的条件是 z 和 $p[z]$是红色的，所以 $p[p[z]]$必存在。根据 $p[z]$是 $p[p[z]]$的左孩子还是右孩子需要考虑 6 种情形，不过其中的 3 种情形与另外 3 种是对称的，这在第 2 行加以确认。仅给出了 $p[z]$是 $p[p[z]]$的左孩子情形的代码，对称的部分代码读者可自行添加。

情形 1：z 的叔叔 y（$=right[p[p[z]]]$）是红色的

图 2-24 展示了情形 1 的状态。当 $p[z]$和 y 都是红色的时候，构成情形 1。由于 $p[p[z]]$是黑色的，可以把 $p[z]$和 y 都着成黑色（第 5、6 行），以此来修正 z 和 $p[z]$都是红色的问题，且将 $p[p[z]]$着成红色（第 7 行），以此保持了性质 5。然后将 $p[p[z]]$作为新的结点 z（第 8 行）重复此 **while** 循环。这样 z 就在树中上升了两层。

图 2-24 显示了过程 RB-INSERT 的情形 1。由于 z 及其父亲 $p[z]$都是红色的，违背了性质 4。无论图 2-24(a)中 z 是右孩子还是图 2-24(b)中 z 是左孩子，都将做一个动作。每一棵子树 α、β、γ、δ 和 ε 都有一个黑色的根，而求具有相同的黑色高度。对情形 1 的代码变换了某些结点的颜色，并保持性质 5：所有从一个定点起向下到叶子的路径都有相同数量的黑色结点。该 **while** 循环将以 z 的祖父 $p[p[z]]$作为新的 z 而继续。现在对性质 4 的违背只可能发生在新的 z 是红色的且其父亲也是红色的。

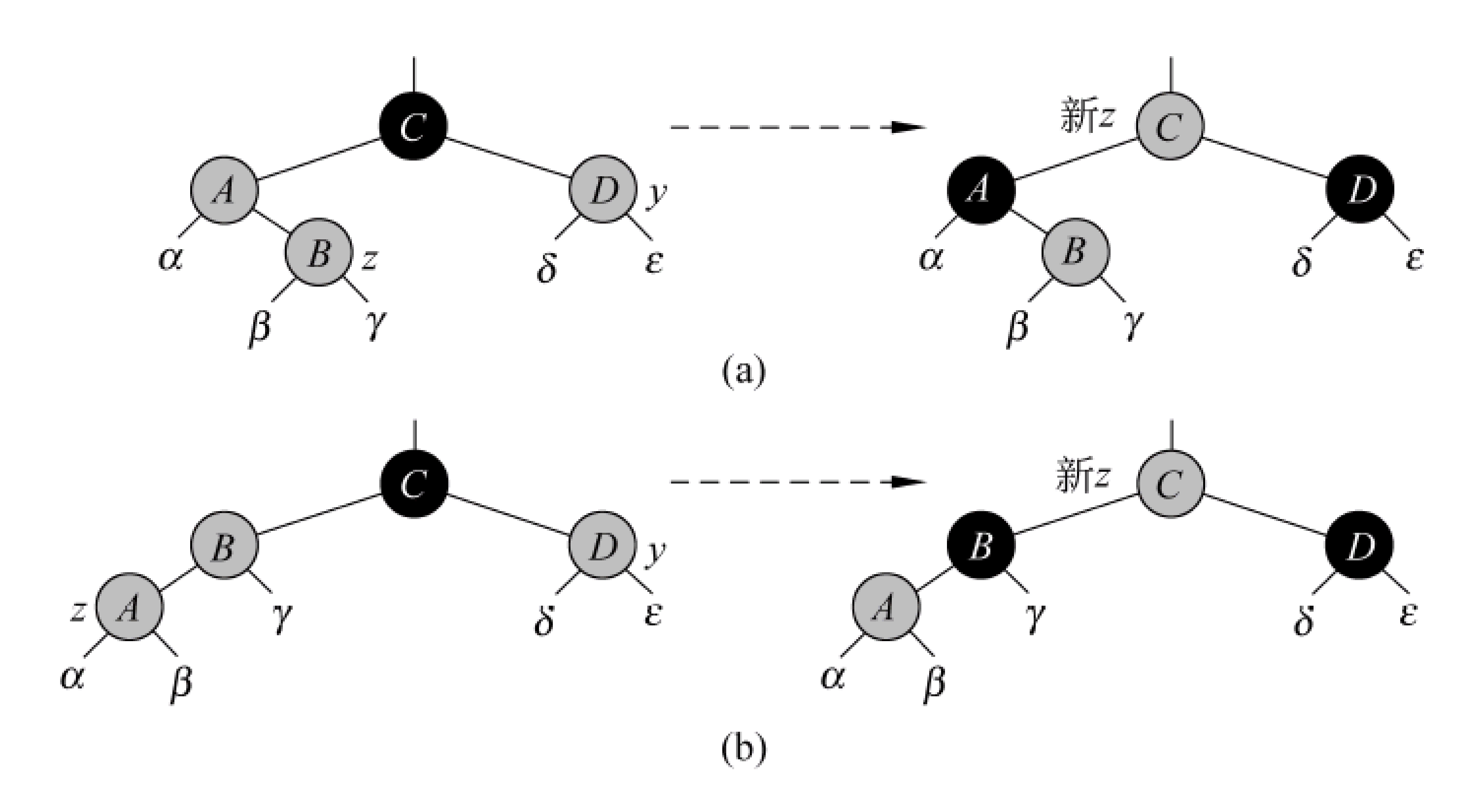


图 2-24　过程 RB-INSERT 的情形 1

情形 2：z 的叔叔是黑色的且 z 是一个右孩子

第 10 行和第 11 行构成情形 2，它在图 2-25 中与情形 3 一起展示。在情形 2 中，结点 z 是其父亲的右孩子。立即利用一次左旋转将其转换成情形 3，在此情形中 z 是一个左孩子。由于 z 和 $p[z]$都是红色的，该旋转既不会影响各结点的黑色高度，也不会违背性质 5。

情形 3：z 的叔叔是黑色的且 z 是一个左孩子

无论是直接进入情形 3 还是通过情形 2 进入情形 3，z 的叔叔 y 是黑色的，否则将执行情形 1。再则，结点 $p[p[z]]$存在，因为上文已经论证过执行了第 2 行和第 3 行此结点就存在了，并在第 10 行将 z 上移一层然后在第 11 行再下移一层，$p[p[z]]$的身份没有改变。在情形 3 中(第 12～14 行)，执行若干颜色改变和一次右旋转，将保持性质 5，然后，由于不再有两个红色结点连在一起，我们就完成了任务。该 **while** 循环不再执行，因为 $p[z]$此时已为黑色。

当循环终止时，$p[z]$是黑色的(若 z 是根，则 $p[z]$是哨兵 $nil[T]$，它是黑色的)，所以循环终止时没有对性质 4 的违背。只可能有对性质 2 的违背。第 16 行恢复了此性质，所以当 RB-INSERT-FIXUP 终止时，所有的红-黑树性质都成立。

图 2-25 所示为过程 RB-INSERT 的情形 2 和情形 3。就像在情形 1 中，性质 4 在情形 2 或情形 3 中被违背，因为 z 及其父亲 $p[z]$都是红色的。每棵子树 α、β、γ 和 δ 都有一个黑色的根(α、β 和 γ 是根据性质 4，而 δ 是因为若否则为情形 1)，以及每一棵都有同样的黑色高度。通过一次左旋转情形 2 可转换成情形 3，它保持了性质 5：从一个结点向下到叶子的所有路径都有相同多的黑色结点。情形 3 做了若干个颜色变化和一次右旋转，这也保持了性质 5。该 **while** 循环就终止，因为性质 4 得以满足：不再有两个红色结点连在一起。

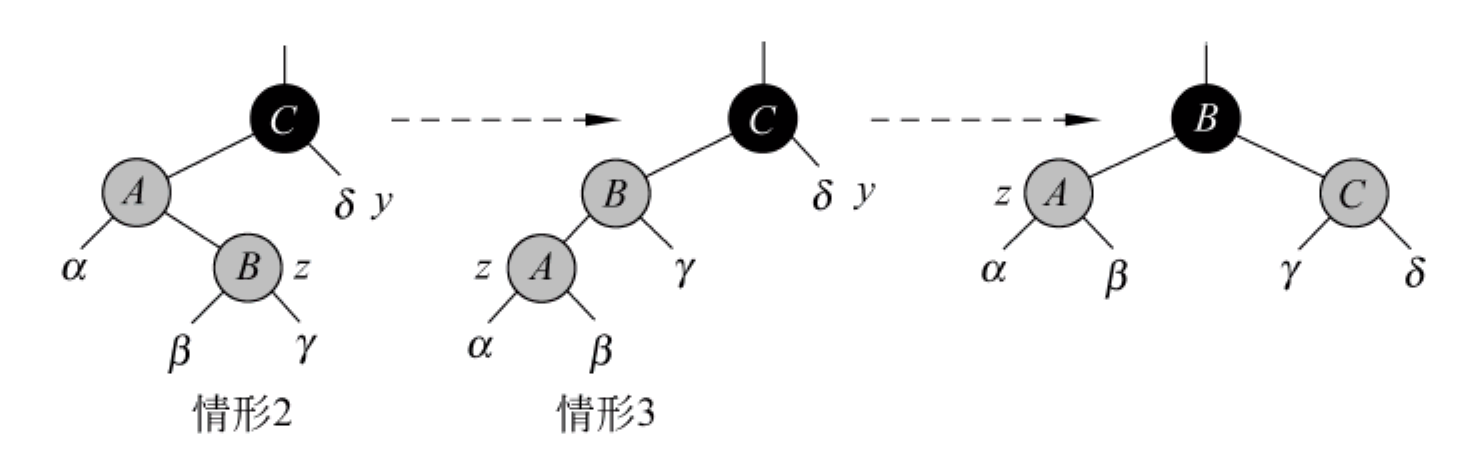


图 2-25　过程 RB-INSERT 的情形 2 和情形 3

RB-INSERT 的运行时间是多少？由于一棵具有 n 个结点的红-黑树的高度是 $O(\lg n)$，RB-INSERT 的第 1～16 行耗时 $O(\lg n)$。在 RB-INSERT-FIXUP 中，**while** 循环只有在情形 1 被执行时才会重复，且指针 z 在树中上移两层。所以该 **while** 循环至多能别执行 $O(\lg n)$次。于是，RB-INSERT 总耗时 $O(\lg n)$。有趣的是旋转至多被执行两次，因为若执行了情形 2 或情形 3，该 **while** 循环就会终止。

3. 删除

和其他对具有 n 个结点的红-黑树的基本操作一样，删除一个结点耗时 $O(\lg n)$。从红-黑树中删除一个结点要比插入一个结点稍稍复杂。

过程 RB-DELETE 是对 TREE-DELETE 过程的一个修改。删除结点后，调用一个辅助过程 RB-DELETE-FIXUP 修改一些颜色并执行若干旋转以恢复红-黑树的性质。

```
RB-DELETE(T,z)
if left[z]=nil[T] or right[z]=nil[T]
   then y←z
   else y←TREE-SUCCESSOR(z)
if left[y]≠nil[T]
   then x←left[y]
```

```
6     else x←right[y]
7 p[x]←p[y]
8 if p[y]=nil[T]
9     then root[T]←x
10    else if y=left[p[y]]
11            then left[p[y]]←x
12            else right[p[y]]←x
13 if y≠z
14    then key[z]←key[y]
15         copy y's satellite data into z
16 if color[y]=BLACK and x≠nil[T]
17    then RB-DELETE-FIXUP(T,x)
18 return y
```

算法 2-23　从红-黑树 T 中删除结点 z 的过程 RB-DELETE

TREE-DELETE 和 RB-DELETE 过程之间有 3 个区别。首先，TREE-DELETE 中所有对 NIL 的引用在 RB-DELETE 中都替换为对哨兵 $nil[T]$的访问。其次，TREE-DELETE 中的第 7 行对 x是否为 NIL 的检测被删除，并在 RB-DELETE 的第 7 行执行无条件赋值 $p[x]\leftarrow p[y]$。于是，若 x 是哨兵 $nil[T]$，其父亲指针指向被删除的结点 y。最后，若 y 是黑色的，在第 16 行和第 17 行调用 RB-DELETE-FIXUP。若 y 是红色的，即使 y 被删除红-黑树性质依然成立，原因如下。

（1）树中的黑色高度没有改变。

（2）没有相邻的红色结点。

（3）由于若 y 是红色的它就不是根，根仍然是黑色的。

传递给 RB-DELETE-FIXUP 的结点 x 是两种结点之一：或者是在 y 未被删除前 y 不是哨兵 $nil[T]$的唯一的孩子，或 y 没有孩子，x 是哨兵 $nil[T]$。在后面的情形中，第 7 行的无条件赋值保证了 x 的父亲现在是 y 的原来的父亲，不管 x 是有关键值的内结点还是哨兵 $nil[T]$，并且无论 y 的颜色如何删除 y 都不会破坏红-黑树的任何性质。而对于后者，当 y 为黑色结点时需要调用过程 RB-DELETE-FIXUP 调整树 T 的结构，恢复可能被破坏的红-黑树性质了。

现在可以来考察过程 RB-DELETE-FIXUP 是如何恢复红-黑树性质的了。

```
RB-DELETE-FIXUP(T,x)
while x≠root[T] and color[x]=BLACK
  do if x=left[p[x]]
       then w←right[p[x]]
         if color[w]=RED
           then color[w]←BLACK              ▷情形 1
             color[p[x]]←RED                ▷情形 1
             LEFT-ROTATE(T,p[x])            ▷情形 1
             w←right[p[x]]                  ▷情形 1
         if color[left[w]]=BLACK and color[right[w]]=BLACK
```

```
10              then color[w]←RED                              ▷情形 2
11                   x←p[x]                                    ▷情形 2
12              else if color[right[w]]=BLACK
13                     then color[left[w]]←BLACK               ▷情形 3
14                          color[w]←RED                       ▷情形 3
15                          RIGHT-ROTATE(T,w)                  ▷情形 3
16                          w←right[p[x]]                      ▷情形 3
17                color[w]←color[p[x]]                         ▷情形 4
18                color[p[x]]←BLACK                            ▷情形 4
19                color[right[w]]←BLACK                        ▷情形 4
20                LEFT-ROTATE(T,p[x])                          ▷情形 4
21                x←root[T]                                    ▷情形 4
22        else (same as then clause with "right" and "left" exchanged)
23 color[x]←BLACK
```

算法 2-24 删除结点后恢复红-黑树性质的过程 RB-DELETE-FIXUP

若 RB-DELETE 删除的结点 y 是黑色的,就会发生 3 个问题。首先,若 y 是根且 y 的一个红色的孩子成为新的根,就违背了性质 2。其次,若 x 和 $p[y]$(现在它也是 $p[x]$)是红色的,这违背了性质 4。再次,y 的删除造成原先任一包含 y 的路径都少了一个黑色结点。这样树中任一 y 的前辈都将违背性质 5。可以通过声称 x 有一份"额外的"黑色来纠正此问题,即若对任一含有 x 的路径都为所含黑色结点数加 1,则在此解释下,性质 5 成立。当删掉 y 后,将其"黑色"性质"推"给它的孩子。现在的问题是 x 既非红色又非黑色,所以违背了性质 1。现在的做法是,x 或为"双黑"或为"红-黑"且分别对包含 x 的路径所含黑色结点数贡献 2 或 1。x 的 *color* 属性仍然或是 RED(若 x 为"红-黑")或是 BLACK(若 x 为"双黑")。换句话说,结点额外的黑色反映在 x 的指针上而非 *color* 属性上。

过程 RB-DELETE-FIXUP 将恢复性质 1、2 和 4。第 1～22 行的 **while** 循环的目标是将树中额外黑色向上移动直至如下。

(1) x 指向一个红-黑结点,在此情形中第 23 行将 x 着成(单一)黑色。

(2) x 指向根,在此情形中可以将额外的黑色直接"移除"。

(3) 可以执行合适的旋转和重新着色。

图 2-26 所示为过程 RB-DELETE-FIXUP 的 **while** 循环中的各种情形。黑色的结点具有 *color* 属性 BLACK,深阴影结点具有 *color* 属性 RED,浅阴影结点具有 *color* 属性 c 或 c',或 RED 或 BLACK。字母 α、β、…、ζ 表示任意子树。在每种情形中,左边的格局通过改变结点的颜色或/并执行一次旋转转换成右边的格局。任一被 x 指向的结点有一份额外的黑色,它或是双黑色或是红-黑色。唯一能使该循环重复的是情形 2。图 2-26(a)是情形 1 通过改变结点 B 和 D 的颜色并执行一次左旋转而转换成情形 2、3 或 4。图 2-26(b)是在情形 2 中,由 x 指针表示的额外黑色通过将结点 D 着成红色并将 x 指向结点 B 而在树中被上移。如果是从情形 1 进入情形 2 的,该 **while** 循环将终止,因为新的结点 x 是红-黑色的,所以其 *color* 属性 c 是 RED。图 2-26(c)是情形 3 通过改变结点 C 和 D 的颜色并执行一次右旋转被转换成情形 4。图 2-26(d)是在情形 4 中,x 表示的额外黑色可以通过改变一些结点的颜色并执行一次左旋转而取消(不违背红-黑树性质),并终止循环。

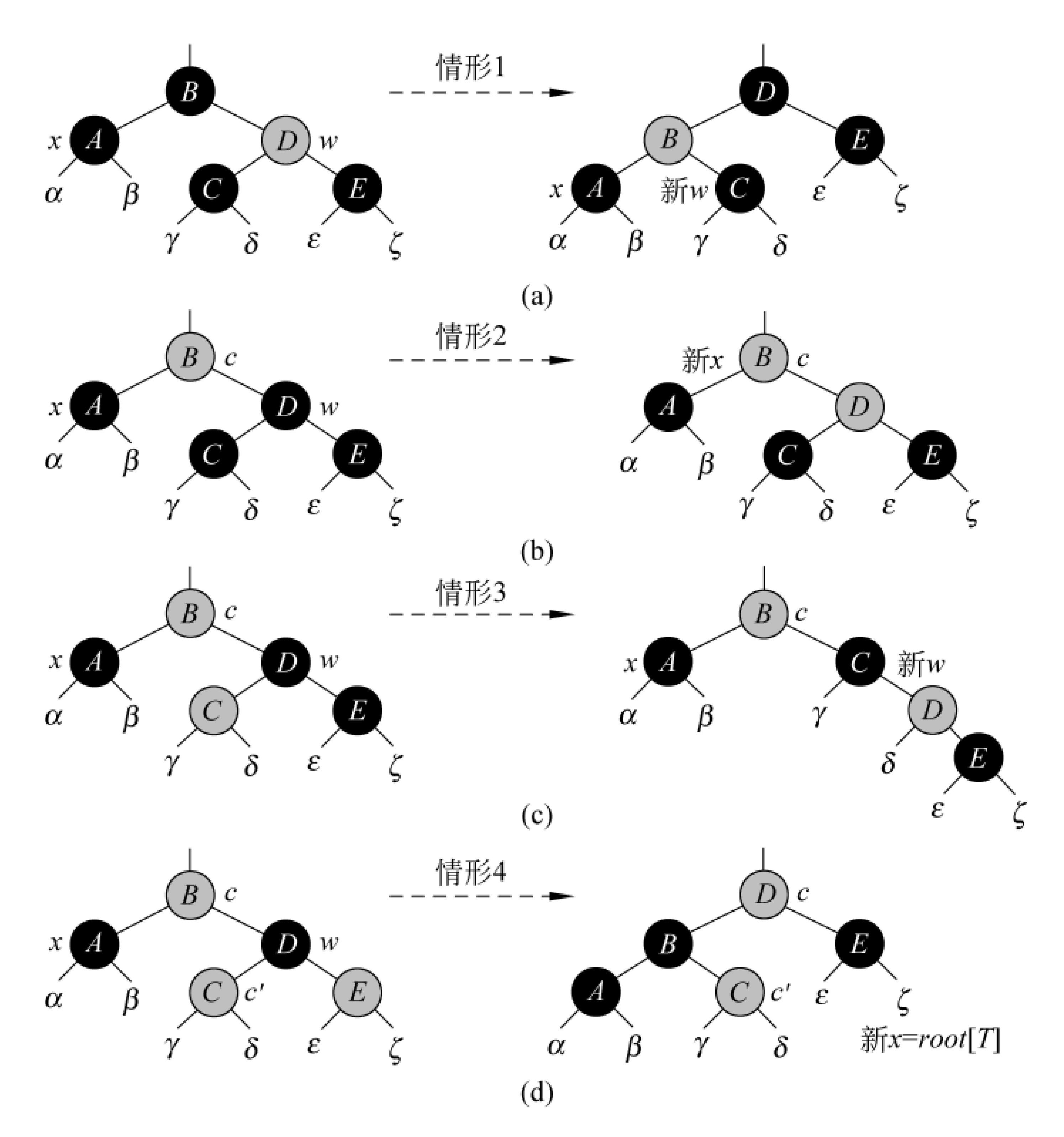


图 2-26　过程 RB-DELETE-FIXUP 的 while 循环中的各种情形

在该 **while** 循环中，x 总是指向非根双黑结点。在第 2 行确认 x 是其父亲 $p[x]$的左孩子还是右孩子(已经给出了 x 是左孩子情形下的代码；x 是右孩子的情形在第 22 行是对称的)。我们用一个指针 w 指向 x 的兄弟。由于结点 x 是双黑的，结点 w 不可能是 $nil[T]$；否则的话，从 $p[x]$到(纯黑色的)叶子 w 的路径所含黑色结点数将小于从 $p[x]$到 x 的路径上的黑色结点数。

代码中的 4 种情况[①]示例于图 2-26。详细地检测每一种情形前，先一般地观察一下如何验证每一种情形所做的转换能保持性质 5。关键思想是在每种情形中，从子树的根(且包含)起到每一棵子树 α、β、…、ζ 的黑色结点数通过转换保持不变。于是，若转换前性质 5 成立，以后将继续成立。例如，在图 2-26(a)中，示例了情形 1，从根起到子树 α 或 β 的黑色结点都是 3，无论是转换前还是转换后(再次提醒，x 附加了一份额外黑色)。同样地，根到 γ、δ、ε 和 ζ 的黑色结点数转换前后也都是 2。在图 2-26(b)中，计数要包括子树根的 $color$ 属性的值 c，它可能是 RED 或 BLACK。若定义 count(RED)＝0 及 count(BLACK)＝1，则从根到 α 的黑色结点数是 2＋count(c)，无论是转换前还是转换后。在此情形中，转换后，新的结点 x 具有 $color$ 属性 c，但此结点真正是红-黑(若 c＝RED)或双黑(若 c＝BLACK)。其他情形可以相仿地进行验证。

① 如在 RB-INSERT-FIXUP 中那样，RB-INSERT-FIXUP 中的几个情形也不是互斥的。

情形 1：*x* 的兄弟 *w* 是红色的

情形 1(RB-DELETE-FIXUP 的第 5～8 行及图 2-26(a))发生在结点 x 的兄弟结点 w 是红色时。由于 w 必有黑色的孩子,可以切换 w 与 $p[x]$的颜色并对 $p[x]$做一次左旋转而不违背任何红-黑树性质。x 的新的兄弟,是旋转前 w 的孩子,现在是黑色的,于是已经将情形 1 转换成了情形 2、3 或 4。

情形 2、3 及 4 发生于 w 是黑色的时,它们与情形 1 的区别就在于 w 的颜色。

情形 2：*x* 的兄弟 *w* 是黑色的,且 *w* 的孩子都是黑色的

在情形 2 中(RB-DELETE-FIXUP 的第 10 行和第 11 行及图 2-26(b)),w 的两个孩子都是黑色的。由于 w 也是黑色的,从 x 和 w 去掉一重黑色,使得 x 仅为黑色而 w 仅为红色,为补偿从 x 和 w 中去除的黑色,把额外的黑色加到 $p[x]$上,它原来可能是红色的也可能是黑色的。通过将 $p[x]$作为新的 x 重复 **while** 循环做到这一点。注意,若从情形 1 进入到情形 2,新的 x 是红-黑色的,这是因为 $p[x]$原来是红色的。所以,新的 x 结点的 $color$ 属性 c 是 RED,当循环检测其循环条件时将终止。新的 x 结点在第 23 行被着成黑色。

情形 3：*x* 的兄弟 *w* 是黑色的,*w* 的左孩子是红色的,而 *w* 的右孩子是黑色的

情形 3(第 13～16 行及图 2-26(c))发生在 w 是黑色的时,其左孩子是红色的而其右孩子是黑色的时。可以切换 w 与其左孩子 $left[w]$的颜色然后对 w 执行一次右旋转而不违背任何红-黑树性质。x 新的兄弟 w 现在是具有红色右孩子的黑结点,于是已经把情形 3 转换成了情形 4。

情形 4：*x* 的兄弟 *w* 是黑色的,而 *w* 的右孩子是红色的

情形 4(第 17～21 行及图 2-26(d))发生在 x 的兄弟 w 是黑色的且 w 的右孩子是红色时。通过做一些结点的颜色变化并对 $p[x]$执行一次左旋转,可以去除 x 上的额外黑色,使它具有纯黑色,而不违背任何红-黑树性质。设置 x 为根,使得 **while** 循环在检测其循环条件时终止。

RB-DELETE 的运行时间如何？由于具有 n 个结点的红-黑树的高度是 $O(\lg n)$,过程中不调用 RB-DELETE-FIXUP 的部分耗时为 $O(\lg n)$。在 RB-DELETE-FIXUP 中,情形 1、3 和 4 执行了常数量的颜色变换及至多 3 次旋转。情形 2 是仅有的让 **while** 循环可能重复的情形。并且指针 x 在树中至多上移 $O(\lg n)$次且不执行旋转操作。于是,过程 RB-DELETE-FIXUP 耗时 $O(\lg n)$并至多执行了 3 次旋转,RB-DELETE 的总运行时间因此为 $O(\lg n)$。

2.4.7 红-黑树的程序实现

1. 数据类型定义

在 C 语言中,把结点类型及红-黑树类型定义成如下的结构体。

```
typedef enum {RED,BLACK} Color;          /* 结点颜色类型 */
typedef struct node{                     /* 二叉树结点类型 */
    struct node *p;                      /* 父结点指针 */
    struct node *left;                   /* 左孩子结点指针 */
    struct node *right;                  /* 右孩子结点指针 */
    Color color;
```

```
7      void  * key;                                                  /* 数据域指针 */
8 }RBNode;
9 RBNode  * creatRBNode(void  * key,int size);                        /* 创建结点 */
10 void clrRBNode(RBNode *  r,void( * proc)(void * ));                /* 清理结点 */
11 typedef struct{                                                    /* 红-黑树类型 */
12     unsigned nodeSize;
13     int ( * comp)(void * ,void * );
14     RBNode  * root;                                                /* 根结点指针 */
15     RBNode  * nil;                                                 /* 哨兵结点指针 */
16 }RBTree;
17 RBTree *  creatRBTree(int size,int( * comp)(void  * ,void  * ));   /* 创建空树 */
18 void clrRBTree(RBTree  * t,void( * proc)(void * ));                /* 清理二叉树 */
19 void inorderRBWalk(RBTree  * t,void ( * proc)(void  * )) ;         /* 中序遍历 */
20 RBNode *  rbMini(RBTree  * t);                                     /* 最小值 */
21 RBNode *  rbMax(RBTree  * t);                                      /* 最大值 */
22 RBNode *  rbSearch(RBTree *  t,void *  k);                         /* 查找 */
23 void rbInsert(RBTree *  t,void  * key);                            /* 插入 */
24 RBNode *  rbDelete(RBTree  * t,RBNode  * x);                       /* 删除 */
```

程序 2-22　定义通用红-黑树类型及声明操作函数的头文件 redblacktree. h

对程序 2-22 的说明如下。

(1) 第 2～8 行的结构体类型 RBNode 定义的是红-黑树结点。与简单的二叉树结点类型 BTreeNode 相比,增加了一个表示结点颜色的属性 color,该属性定义成第 1 行的枚举类型 Color。由于将数据域 key 定义成 **void** * ,可以指向存储任何类型的内存单元,所以是通用的数据结构。第 9 行和第 10 行声明的函数 creatRBNode、clrRBNode,分别用来创建结点、清理结点空间。

(2) 第 11～16 行的结构体类型 RBTree 定义的是红-黑树。它有 4 个属性:结点数据域的存储宽度 nodeSize、结点数据比较规则 comp、根结点指针 root 和哨兵结点指针 nil。第 17 行和第 18 行声明的函数 creatRBTree 和 clrRBTree 是用来创建空红-黑树和清理红-黑树空间的。第 19 行声明的函数 inorderRBWalk 用来对红-黑树进行中序遍历。

(3) 第 20 行和第 21 行声明的函数 rbMini 和 rbMax 分别计算红-黑树中关键值最小/最大的结点。第 22 行声明的函数 rbSearch 实现算法 2-14 中的过程 TREE-SEARCH,这是因为红-黑树本质上就是一棵二叉搜索树。第 23 行和第 24 行声明的函数 rbInsert 和 rbDelete 分别实现算法 2-21 的 RB-INSERT 过程和算法 2-23 的 RB-DELETE 过程。

下面来分别定义上述函数。

2. 结点维护

二叉树是由结点根据父子关系连接起来构成的。结点是二叉树的"基础设施"。需要对基础设施进行常规的维护,包括创建结点和清理不再使用的结点的存储空间。

```
RBNode  * creatRBNode(void  * key,int size) {            /* 创建结点 */
    RBNode  * x=(RBNode * ) malloc(sizeof (RBNode));
    x->key=(void * ) malloc(size);
```

```
    memcpy(x->key,key,size);
    x->p=x->left=x->right=&nil;
    x->color=RED;
   return x;
}
void clrRBNode(RBNode * r,void( * proc)(void * )) {        /* 清理二叉树结点 */
    r->p=r->left=r->right=NULL;
   if(proc)
        proc(r->key);
   free(r->key);
}
```

程序 2-23　对红-黑树结点进行维护的 C 函数

由于红-黑树本质上就是二叉树,所以其结点仅比二叉树结点多一个表示颜色的属性 color。相应地,对结点的维护操作中,创建结点时需要设置颜色值,默认情况下,设为红色(RED)。其他的维护操作与对普通二叉树结点的完全一致。

3. 红-黑树常规维护

对红-黑树的常规维护操作包括创建红-黑树、清理不再使用的红-黑树的存储空间以及对红-黑树的遍历。

```
static RBNode nil={NULL,NULL,NULL,BLACK,NULL};                    /* 哨兵结点 */
RBTree * creatRBTree(int size,int( * comp)(void * ,void * )) {    /* 创建空红-黑树 */
   RBTree * t=(RBTree * )malloc(sizeof(RBTree));                  /* 分配空间 */
   t->nodeSize=size;                                              /* 设置结点数据存储宽度 */
   t->comp=comp;                                                  /* 设置结点数据比较规则 */
   t->nil=&nil;                                                   /* 设置哨兵结点 */
   t->root=t->nil;                                                /* 将树置空 */
   return t;
}
static void clrTree(RBNode * r,void( * proc)(void * )) {          /* 清理二叉树 */
   if(r==&nil)
        return;
   if (r->left != &nil)                                           /* 清理左子树 */
        clrTree(r->left,proc);
   if (r->right != &nil)                                          /* 清理右子树 */
        clrTree(r->right,proc);
   clrRBNode(r,proc);                                             /* 清理本结点 */
   free(r);
}
void clrRBTree(RBTree * t,void( * proc)(void * )){
   clrTree(t->root,proc);
}
static void inorder(RBNode * r,void( * proc)(void * )) {    /* 对以 r 为根的二叉树中序遍历 */
   if(r->left!=&nil)                                              /* 遍历左子树 */
```

```
25         inorder(r->left,proc);
26     proc(r->key);                                        /* 处理根结点数据 */
27     if(r->right!=&nil)                                   /* 遍历右子树 */
28         inorder(r->right,proc);
29 }
30 void inorderRBWalk(RBTree *t,void (*proc)(void *)){    /* 对红-黑树进行中序遍历 */
31     if(t->root==t->nil)
32         return;
33     inorder(t->root,proc);
34 }
```

程序 2-24　红-黑树常规维护函数

对程序 2-24 的说明如下。

(1) 为节省存储空间,第 1 行定义一个全局量——哑结点 nil。它作为任何一棵红-黑树的哨兵结点。根据哨兵结点的特性,nil 的所有指针域包括父指针、孩子指针及关键值指针均指向空(NULL),颜色域置为黑色(BLACK)。利用此全局哑结点 nil,第 2～9 行的 creatRBTree 函数对创建的红-黑树设置其哨兵结点,并用传递给它的参数 size 和 comp 设置红-黑树结点数据存储宽度和结点数据比较规则。

(2) 由于将红-黑树定义成由根结点指引并具有哨兵结点、结点存储宽度、结点数据比较规则的结构体,与将树仅视为结点的父子链接不同,不能直接对指向一棵红-黑树的指针运用递归操作。所以,第 20～22 行定义的清理参数 t 指定的红-黑树存储空间的函数 clrRBTree,需要调用第 10～19 行定义的清理参数 r 指定的结点作为根的二叉树空间的递归函数 clrTree。细心的读者也许已经看出来,函数 clrTree 的操作与程序 2-20 定义的二叉树空间清理函数 clrBTree 是一样的。有一点不同的是 clrTree 定义成 **static** 函数,这样就将其可用范围局限在本文件中,作为 clrRBTree 调用的功能函数而对外部加以屏蔽。

(3) 与(2)中对参数类型所决定的函数 clrRBTree 与 clrTree 定义方式的变化一样的理由,第 30～34 行定义的对由参数 t 指定的红-黑树做中序遍历操作的函数需调用第 23～29 行定义的对由参数 r 指定的结点为根的二叉树做中序遍历的递归函数 inorder。inorder 也是 **static** 函数。

4. 红-黑树的查询操作函数

对红-黑树的查询操作包括在其中查找指定关键值结点、计算树中最小/最大关键值结点等。

```
RBNode* rbSearch(RBTree* t,void* k){
    int contrast;
    RBNode *r=t->root;
    while(r!=t->nil&&(contrast=t->comp(r->key,k))){
        if(contrast>0)
            r=r->left;
        else
            r=r->right;
```

```
9        }
10     if(r==t->nil)
11         return &nil;
12     return r;
13 }
14 static RBNode* treeMini(RBNode *r){          /* 求最小值结点 */
15     while(r->left!=&nil)                     /* 找到左子树中最左边的结点 */
16         r=r->left;
17     return r;
18 }
19 RBNode* rbMini(RBTree *t){
20     if(t->root==t->nil)
21         return NULL;
22     return treeMini(t->root);
23 }
24 static RBNode* treeMax(RBNode *r){           /* 求最大值结点 */
25     while(r->right!=&nil)                    /* 找到右子树中最右边的结点 */
26         r=r->right;
27     return r;
28 }
29 RBNode* rbMax(RBTree *t){
30     if(t->root==t->nil)
31         return NULL;
32     return treeMax(t->root);
33 }
```

程序 2-25　红-黑树查询操作函数的定义

对程序 2-25 的说明如下。

(1)第 1～13 行定义的函数 rbSearch 实际上实现的是算法 2-14 的关于二叉搜索树的查找过程 TREE-SEARCH。虽然参数改成了红-黑树 t,但在函数内部查找起点是从 t 的根结点开始的。这是因为,红-黑树本质上就是一棵二叉搜索树。程序代码与算法的伪代码结构非常接近,读者可对照研读。

(2) 第 19～23 行定义的函数 rbMini 计算并返回红-黑树 t 中关键值最小的结点。由于对参数传递进来的红-黑树 t 不便递归,所以调用第 14～18 行定义的函数 treeMini,从 t 的根结点起递归地进行计算。函数 treeMini 实现的是算法 2-15 的 TREE-MINIMUM 过程,代码几乎一致。将 treeMini 定义成 **static** 函数的目的是向外界屏蔽该函数。

(3) 与(2)相似,第 29～33 行定义的函数 rbMax 计算并返回红-黑树 t 中关键值最大的结点。该函数调用第 24～28 行定义的实现算法 2-16 的 TREE-MAXIMUM 过程的递归函数 treeMax。注意,treeMax 也是一个 **static** 函数。

5. 红-黑树的旋转

红-黑树的旋转操作是保持红-黑树性质进而维护红-黑树平衡的基本操作。旋转操作包括左旋转和右旋转。

```
1 static void leftRotate(RBTree *t,RBNode *x){           /*树 t 中以结点 x 为轴左旋转*/
2     RBNode *y=x->right;                                 /*设置 y*/
3     x->right=y->left;                                   /*将 y 的左子树设置为 x 的右子树*/
4     y->left->p=x;
5     y->p=x->p;                                          /*将 x 链接到 y 的父结点*/
6     if(x->p==t->nil)
7         t->root=y;
8     else if(x==x->p->left)
9         x->p->left=y;
10    else
11        x->p->right=y;
12    y->left=x;                                          /*将 x 置为 y 的左孩子*/
13    x->p=y;
14 }
15 static void rightRotate(RBTree *t,RBNode *x){
16    RBNode *y=x->left;                                  /*设置 y*/
17    x->left=y->right;                                   /*将 y 的右子树设置为 x 的左子树*/
18    y->right->p=x;
19    y->p=x->p;                                          /*将 x 链接到 y 的父结点*/
20    if(x->p==t->nil)
21        t->root=y;
22    else if(x==x->p->right)
23        x->p->right=y;
24    else
25        x->p->left=y;
26    y->right=x;                                         /*将 x 置为 y 的右孩子*/
27    x->p=y;
28 }
```

程序 2-26　实现红-黑树旋转操作的 C 函数

对程序 2-26 的说明如下。

(1) 第 1～14 行定义的函数 leftRotate 实现算法 2-20 的 LEFT-ROTATE 过程。在红-黑树 t 中绕结点 x 做左旋转。该函数无论是参数还是代码结构都几乎是算法伪代码过程的翻版，两者仅有对对象的属性表示方式有所不同。读者可对照研读。

(2) 第 15～28 行定义的函数 rightRotate 在红-黑树 t 中绕结点 x 做右旋转。其代码与 leftRotate 的代码是对称的，即所操作的结点的左右孩子互换。读者也可对照研读。

(3) 函数 leftRotate 和 rightRotate 都定义成 **static** 函数，目的就是要向外界屏蔽这两个函数，因为它们是在红-黑树中插入或删除结点的操作中需要调用的操作步骤，仅此而已，不做他用。

6. 在红-黑树中插入结点

利用红-黑树的旋转操作，在红-黑树中进行插入结点，可保持树的平衡性。下列是实现在红-黑树中插入元素的算法 2-21 和算法 2-22 的 C 函数。

```
static void insertFixup(RBTree *t,RBNode *z){          /* 插入调整 */
    RBNode *y;
    while(z->p->color==RED)                            /* 只要 z 和 z 的父结点均为红色 */
        if(z->p==z->p->p->left){
            y=z->p->p->right;
            if(y->color==RED){
                z->p->color=BLACK;
                y->color=BLACK;
                z->p->p->color=RED;
                z=z->p->p;
            }else{
                if(z==z->p->right){
                    z=z->p;
                    leftRotate(t,z);
                }
                z->p->color=BLACK;
                z->p->p->color=RED;
                rightRotate(t,z->p->p);
            }
        }else{
            y=z->p->p->left;
            if(y->color==RED){
                z->p->color=BLACK;
                y->color=BLACK;
                z->p->p->color=RED;
                z=z->p->p;
            }else{
                if(z==z->p->left){
                    z=z->p;
                    rightRotate(t,z);
                }
                z->p->color=BLACK;
                z->p->p->color=RED;
                leftRotate(t,z->p->p);
            }
        }
    t->root->color=BLACK;
}
void rbInsert(RBTree* t,void *data){
    int (*comp)(void*,void*)=t->comp;
    int size=t->nodeSize;
    RBNode *x=t->root, *y=t->nil, *z=creatRBNode(data,size);
    while(x!=t->nil){                                  /* 确定 z 的父结点 */
        y=x;                                           /* y 指向下一轮 x 的父结点 */
        if(comp(z->key,x->key)<0)
```

```
46              x=x->left;
47          else
48              x=x->right;
49      }
50      z->p=y;                                  /* z 是 y 的孩子 */
51      if(y==t->nil)                            /* t 是一棵空树 */
52          t->root=z;
53      else if(comp(z->key,y->key)<0)           /* 新结点的关键值小于父结点的关键值 */
54          y->left=z;
55      else
56      y->right=z;
57      z->left=t->nil;
58      z->right=t->nil;
59      z->color=RED;
60      insertFixup(t,z);
61 }
```

程序 2-27　实现算法 2-21、算法 2-22 的 C 函数定义

对程序 2-27 的说明如下。

(1) 第 1～38 行定义的函数 insertFixup 实现算法 2-22 的 RB-INSERT-FIXUP 过程。其中，第 4～20 行的 **if** 语句的第一个分句对应于算法 2-22 的第 2～15 行的 **if** 语句部分。这一部分的代码结构十分接近，程序中第 21～38 行的 **else** 分句部分是与前一部分对称的代码，即将前者中的 left、right 互换，且将 leftRotate 与 rightRotate 互换得到代码。

(2) 第 39～61 行定义的函数 rbInsert 实现的是算法 2-21 的 RB-INSERT 过程。与算法过程稍有不同的是，第 2 个参数传递的不是要插入的结点 z，而是结点中的数据 data。所以，在函数体内的第 42 行用 data 作为参数调用结点生成函数 creatRBNode 创建新结点 z。为使代码表示简洁，将树 t 的 nodeSize 域设置为变量 size，而将 t 的比较规则设置为函数指针 comp。函数与算法的代码结构十分接近。

7. 在红-黑树中删除结点

和在红-黑树中插入结点相仿，在红-黑树中删除结点也需要调用树的旋转操作来保持平衡。此外，为了找到删除结点的后继，进行结点间值的调整还需要实现算法 2-17 中的计算二叉搜索树中指定结点的后继结点的 TREE -SUCCESSOR 过程。

```
RBNode *  treeSuccessor(RBNode * r){           /* 求结点 r 在树中的后继 */
    RBNode * y;
    if(r->right!=&nil)                          /* 若右子树非空 */
        return treeMini(r->right);              /* 后继是右子树中的最小值结点 */
    y=r->p;
    while(y!=&nil&&r==y->right){/**/
        r=y;
        y=y->p;
    }
```

```
    return y;
}
static void deleteFixup(RBTree * t,RBNode * x){
    RBNode * w;
    while((x!=t->root)&&(x->color==BLACK))
        if(x==(x->p->left)){
            w=x->p->right;
            if(w->color==RED){
                w->color=BLACK;
                x->p->color=RED;
                leftRotate(t,x->p);
                w=x->p->right;
            }
            if((w->left->color==BLACK)&&(w->right->color==BLACK)){
                w->color=RED;
                x=x->p;
            }else{
                if(w->right->color==BLACK){
                    w->left->color=BLACK;
                    w->color=RED;
                    rightRotate(t,w);
                    w=x->p->right;
                }
                w->color=x->p->color;
                x->p->color=BLACK;
                if(w!=t->nil)
                    w->right->color=BLACK;
                leftRotate(t,x->p);
                x=t->root;
            }
        }else{
            w=x->p->left;
            if(w->color==RED){
                w->color=BLACK;
                x->p->color=RED;
                rightRotate(t,x->p);
                w=x->p->left;
            }
            if(w->right->color==BLACK&&w->left->color==BLACK){
                w->color=RED;
                x=x->p;
            }else{
                if(w->left->color==BLACK){
                    w->right->color=BLACK;
                    w->color=RED;
                    leftRotate(t,w);
                    w=x->p->left;
```

```
57                      }
58                      w->color=x->p->color;
59                      x->p->color=BLACK;
60                      if(w!=t->nil)
61                          w->left->color=BLACK;
62                      rightRotate(t,x->p);
63                      x=t->root;
64                  }
65              }
66      x->color=BLACK;
67  }
68 RBNode * rbDelete(RBTree *t,RBNode *z){
69      RBNode *y,*x;
70      if(z->left==t->nil || z->right==t->nil)
71          y=z;
72      else
73          y=treeSuccessor(z);
74      if(y->left!=t->nil)
75          x=y->left;
76      else
77          x=y->right;
78      x->p=y->p;
79      if(y->p==t->nil)
80          t->root=x;
81      else if(y==y->p->left)
82          y->p->left=x;
83      else
84          y->p->right=x;
85      if(y!=z)
86          memcpy(z->key,y->key,t->nodeSize);
87      if(y->color==BLACK&&x!=t->nil)
88         deleteFixup(t,x);
89      return y;
90 }
```

程序 2-28　定义实现红-黑树结点删除操作的 C 函数

对程序 2-28 的说明如下。

(1) 第 1～11 行定义的函数 treeSuccessor 实现算法 2-17 的 TREE-SUCCESSOR 过程，计算由参数 r 指定的结点在二叉搜索树中的后继结点。代码结构与伪代码相同，不再赘述。它仅用于红-黑树删除操作中恢复红-黑树性质时需要调用。

(2) 第 12～67 行定义的函数 deleteFixup 实现算法 2-24 的 RB-DELETE-FIXUP 过程，供在第 68～90 行定义的函数 rbDelete 调用，删除结点后恢复红-黑树性质。其中，第 15～40 行的 **if** 语句的第一个分句对应算法过程中第 2～21 行的 **if** 语句部分。第 40～65 行的 **else** 分句与 **if** 分句是对称的，即将前者中的 left、right 互换，且将 leftRotate 与 rightRotate 互换得到代码。

(3) 第 68～90 行定义的函数 rbDelete 实现的是算法 2-21 的 RB-DELETE 过程。程序代码与伪代码几乎一一对应。

为便于代码重用,将程序 2-23～程序 2-28 的代码保存为文件夹 datastructure 中的源文件 redblacktree.c。

2.4.8 二叉搜索树的应用

二叉搜索树作为表示动态集合的数据结构,其时间效率比链表好。下列是用红-黑树代替链表解决“数据规范”问题的 C 程序。为节省篇幅,仅列出对整型数据处理的函数。

```
1 char *intProccess(char *s){                                    /*整型数据处理函数*/
2     StrInputStream ssin;                                       /*创建串输入流*/
3     RBTree *T=creatRBTree(sizeof(int),intGreater);             /*用树 T 存放处理后的数据*/
4     int x;
5     initStrInputStream(&ssin,s);
6     initStrOutputStream(&ssout,250);
7     while(!sisEof(&ssin)){                                     /*处理数据*/
8         readInt(&ssin,&x);
9         if(rbSearch(T,&x)==T->nil)                             /*若 x 没有出现在树 T 中*/
10            rbInsert(T,&x);
11    }
12    inorderRBWalk(T,putInt);
13    clrRBTree(T,NULL);
14    writeString(&ssout,"\n");
15    return ssout.begin;
16 }
```

程序 2-29 解决“规范数据”问题的程序 2-3 的二叉搜索树版本 normalize1.c 片段

与程序 2-9 相比,第 9 行和第 10 行的 **if** 语句检测到当前读到数据项 x 未在处理过的数据集合 T(表示为二叉搜索树的全序集)中出现过,直接将 x 插入 T 中,而无须像链表那样先找到插入位置,再执行插入操作就能得到有序结果。

2.5 散列表

2.5.1 直接寻址表与散列表

1. 直接寻址表

当关键值的全集 U 足够小时直接寻址是一项简单技术。假定某应用需要一个动态集合,其关键值取自全集 $U=\{0,1,\cdots,m-1\}$,其中的 m 不是太大。假定没有两个元素具有相同的关键值。

利用一个称为**直接寻址表**的数组 $T[0..m-1]$来表示该动态集合，其中的每一个元素称为一个槽位，对应于全集 U 中的一个关键值。图 2-27 示例了该方法；槽位 k 指向集合中关键值为 k 的元素。若集合中不包含关键值为 k 的元素，则 $T[k]=$NIL。

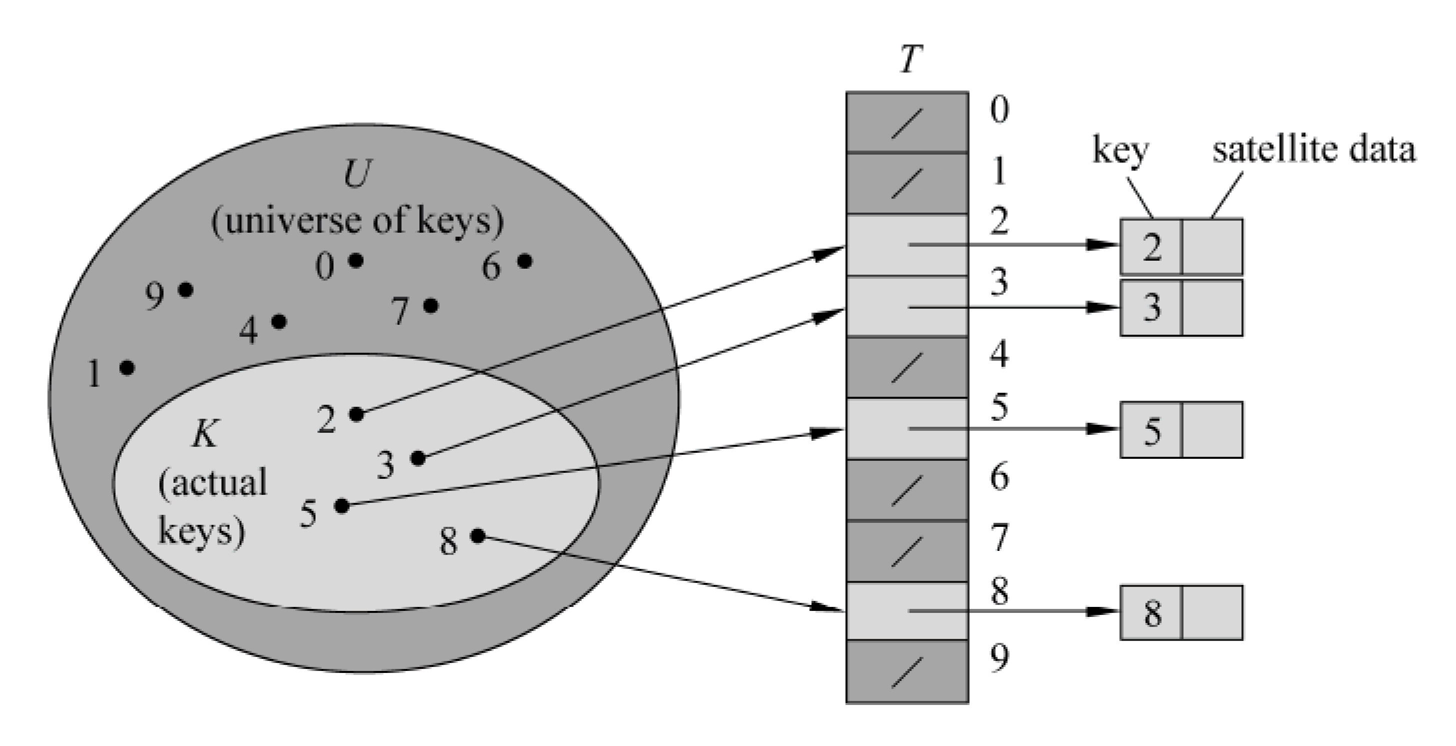


图 2-27　利用直接寻址表 T 实现一个动态集合

图 2-27 显示了利用直接寻址表 T 实现一个动态集合。全集 $U=\{0,1,\cdots,9\}$中的每一个关键值对应表中的一个下标。实际关键值集合 $K=\{2,3,5,8\}$确定了表中含有指向各元素的指针的槽位。深色阴影的其他槽位含有 NIL。

对直接寻址表 T 实现字典操作是轻而易举的。

```
DIRECT-ADDRESS-SEARCH(T,k)
return T [k]
DIRECT-ADDRESS-INSERT(T,x)
T[key[x]]←x
DIRECT-ADDRESS-DELETE(T,x)
T[key[x]]←NIL
```

算法 2-25　对直接寻址表的字典操作过程

每一个操作都很快，只需 $O(1)$时间。

2. 散列表

直接寻址的困难之处是显而易见的：若全集 U 很大时，在通常的计算机中对给定的内存容量存储规模为$|U|$的表 T 是不现实的，甚至是不可能的。此外，实际存储的关键值集合 K 可能相对于 U 来说很小，以至于 T 中的大多数槽位会被浪费。

当存储在字典中的关键值集合 K 比所有可能关键值的全集 U 小得多时，一个散列表所需的空间远比一个直接寻址表的要少得多。具体地说，所需的空间可以节省为 $\Theta(|K|)$，并且在散列表中查找一个元素仍保持 $O(1)$的时间。唯一需要说明的是此上界是对平均时间而言，而对直接寻址表来说却是最坏情形时间的上界。

在直接寻址方法中，一个关键值为 k 的元素存储在槽位 k 处。在散列方法中，此元素存储在槽位 $h(k)$中；也就是说，使用一个**散列函数** h 根据关键值 k 计算出槽位。此处的 h 把全集 U 映射到**散列表** $T[0..m-1]$的槽位：

$$h:U \rightarrow \{0,1,\cdots,m-1\}$$

图 2-28 显示了利用散列函数 h 将关键值映射到散列表的槽位。关键值 k_2 和 k_5 映射到相同的槽位,所以它们冲突了。

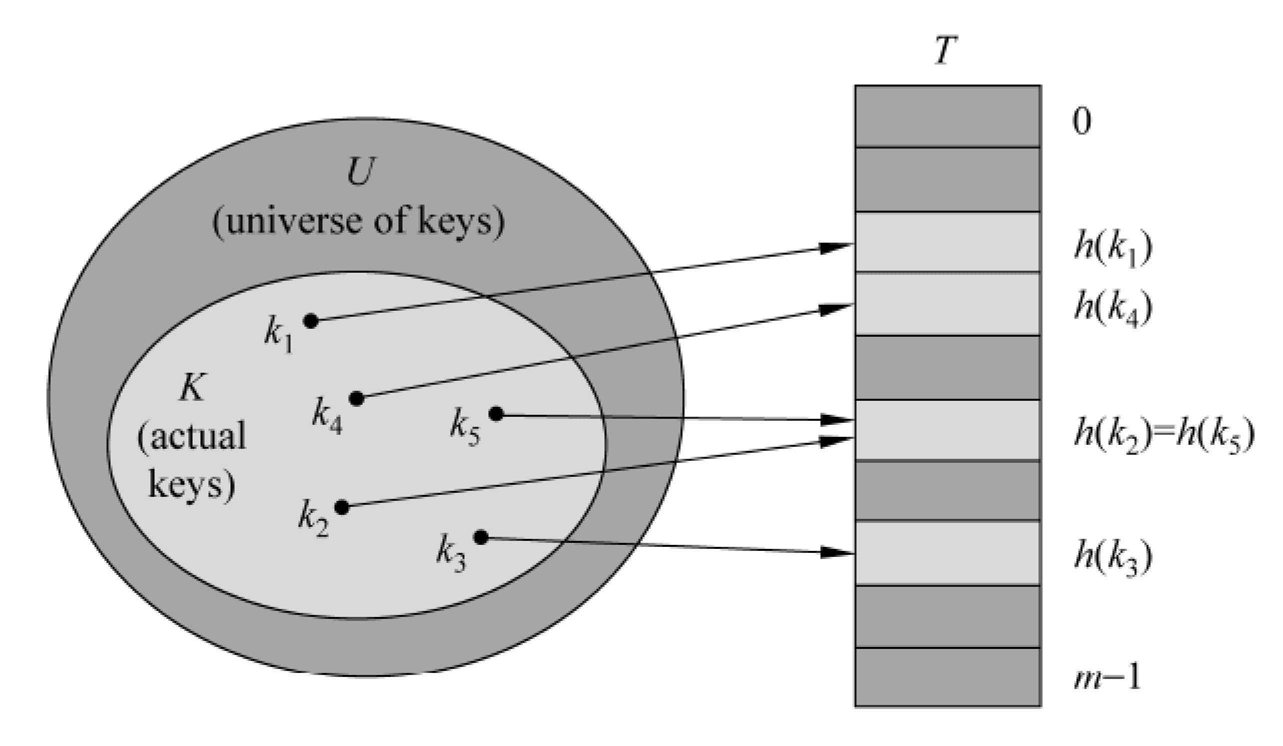


图 2-28　利用散列函数 h 将关键值映射到散列表的槽位

我们说具有关键值 k 的元素散列到槽位 $h(k)$,也说 $h(k)$是关键值 k 的散列值。图 2-28 示例了此基本思想。散列函数的要点是缩小了要处理的数组下标范围。不是$|U|$个值而只需 m 个值。所需的存储空间也相应地减少了。

在创建散列函数的**除法散列法**中,通过计算 k 除以 m 的余数将关键值 k 映射到 m 个槽位中的一个,即散列函数为

$$h(k) = k \bmod m$$

例如,若散列表的大小为 $m=12$ 且 $k=100$,则 $h(k)=4$。由于仅需要做一次除法操作,用除法方法进行散列是很快的。

这样做存在着一个问题:两个关键值可能散列到同一个槽位,把此状况称为一个**冲突**。

2.5.2　用拉链法解决冲突

1. 拉链法散列表的字典操作

在**拉链法**中,把所有散列到同一槽位中的元素放到一个链表中,如图 2-29 所示。槽位 j 包含一个指向有所有散列到 j 的元素构成的链表的指针;若没有这样的元素则槽位 j 包含 NIL。

图 2-29 所示为利用拉链解决冲突。散列表的每个槽位 $T[j]$含有一个由所有散列到 j 的关键值组成的链表。例如,$h(k_1)=h(k_4)$ 和 $h(k_5)=h(k_2)=h(k_7)$。

设散列函数为 h,若用拉链法解决冲突,对一个散列表作的字典操作是很容易实现的。

```
CHAINED-HASH-SEARCH(T,k)                 ▷在散列表 T 中查找关键值为 k 的元素
return LIST-SEARCH(T[h(k)],k)
CHAINED-HASH-INSERT(T,x)                 ▷在散列表 T 中插入元素 x
if CHAINED-HASH-SEARCH(T,key[x])=NIL
    then 调用 LIST-INSERT 将 x 插入到表 T[h(key[x])]的表首
CHAINED-HASH-DELETE(T,k)                 ▷在散列表 T 中删除关键值为 k 的元素
```

```
x←LIST-SEARCH (T[h(k)],k)
LIST-DELETE(T[h(k)],x)
```

算法 2-26　散列表的查找、插入与删除过程

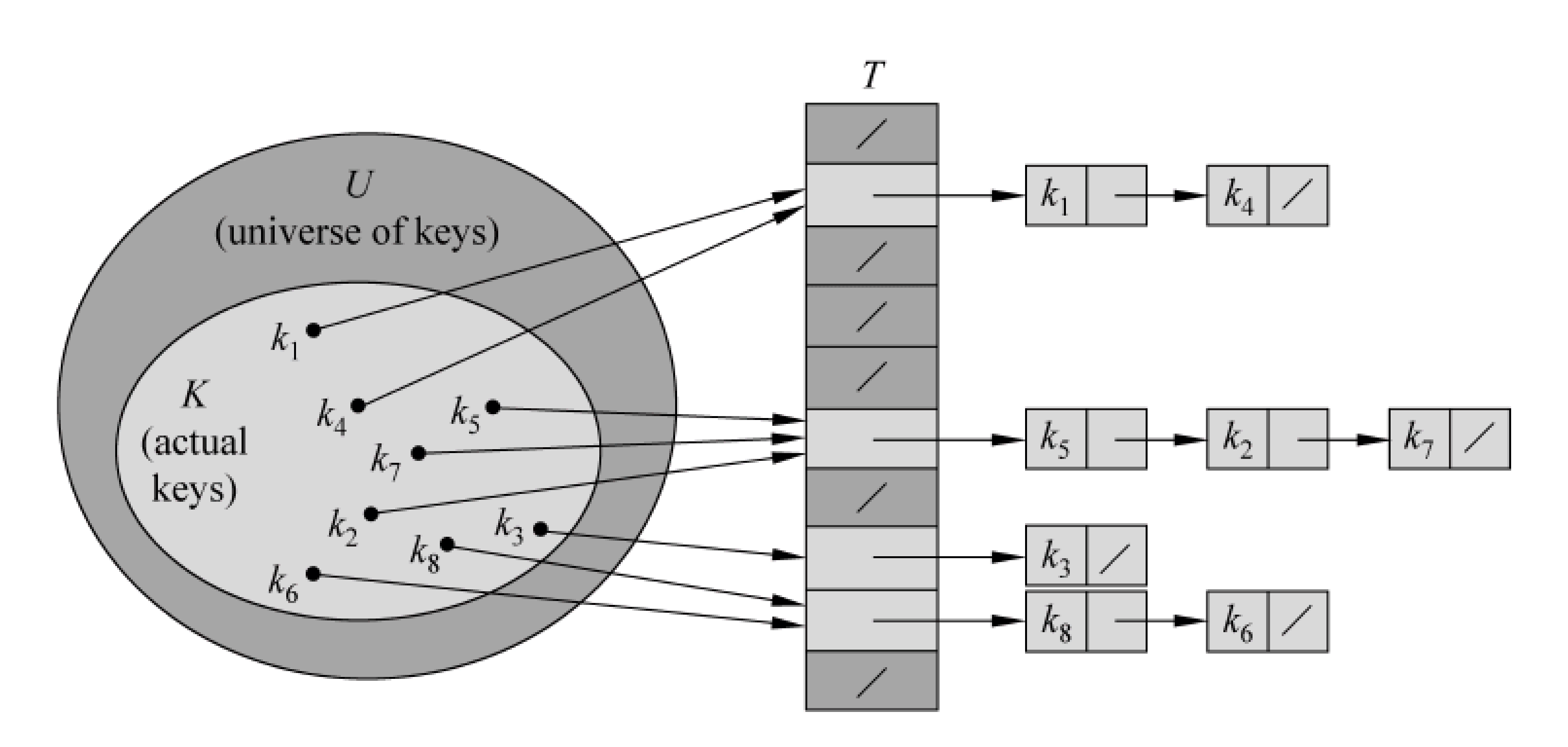


图 2-29　利用拉链解决冲突

无论是在散列表中插入元素的 CHAINED-HASH-INSERT 过程，还是从散列表中删除元素的 CHAINED-HASH-DELETE 过程，都需要先在表中进行一次查找。对于前者来说，查找的目的是确定表中是否已经存在与指定元素 x 的关键值相同的元素，只有当 x 的关键值未曾出现在表中，才将其插入到表中，以此保证表中元素的关键值两两不同。而对于后者来说，需要确定表中是否存在与指定关键值 k 相等的元素，然后再将其从关键值对应的槽位链表中删除。由于表元素是存储在槽位链表中的，所以在过程 CHAINED-HASH-DELETE 中，调用的是链表查找过程，它返回的是链表结点，这样才能正确地删除元素。

2. 用拉链法的散列技术分析

用拉链法的散列技术能做得多好呢？特别地，用一个给定的关键值查找元素要花费多长时间呢？

给定一个具有 m 个槽位、存储了 n 个元素的散列表 T，定义 T 的**装载因子** α 为 n/m，即一个拉链表中的平均元素数目。我们的分析将用 α 来表示，α 可能小于、等于或大于 1。

使用拉链法的散列技术在最坏情形下的表现是很糟糕的：所有 n 个关键值都散列到同一个槽位，构成了一个长度为 n 的链表。最坏情形下的查找耗时 $\Theta(n)$ 加上计算散列函数的时间——并不比就使用一个链表来存储所有元素更好。显然，散列表不会用到最坏情形下的性能。

散列法的平均性能取决于散列函数 h 如何将关键值集合均匀地映射到 m 个存储槽位。现在假定任一给定的元素是等概地散列到 m 个槽位中的任何一个，而与其他元素被散列到哪个槽位无关，把这一假定称为**简单均匀散列**。

对 $j=0,1,\cdots,m-1$，用 n_j 来表示链表 $T[j]$ 的长度，所以：

$$n = n_0 + n_1 + \cdots + n_{m-1} \tag{2-1}$$

且 n_j 的平均值为 $E[n_j]=\alpha=n/m$。

假定散列值 $h(k)$ 可以在 $O(1)$ 时间内算得，所以查找一个具有关键值 k 的元素所需的时间线性地依赖于链表 $T[h(k)]$ 的长度 $n_{h(k)}$。先不管计算散列函数及访问槽位 $h(k)$ 所需

的 $O(1)$时间,考虑查找算法检测过的元素个数的期望值,即在链表 $T[h(k)]$中为检测关键值是否为 k 而考察的元素个数。仅考虑查找不成功的情形,即散列表中没有关键值为 k 的元素。查找成功所用的时间一定不会多于查找不成功所用的时间。

引理 2-3 在一个利用拉链法解决冲突的散列表中,在简单均匀散列的假定下,不成功查找的期望时间是 $\Theta(1+\alpha)$。

证明 在简单均匀散列假设下,任一尚未存储到散列表中的关键值 k 将被等概率地散列到 m 个槽位中的任何一个。对一个关键值 k 的不成功查找的期望时间是搜索到链表 $T[h(k)]$的表尾的期望时间为 $\Theta(1+\alpha)$,该拉链表的期望长度 $E[n_{h(k)}]=\alpha$。于是,在一次不成功查找中期望搜索到的元素个数为 α,且所需的总的时间(包括计算 $h(k)$的时间)为 $\Theta(1+\alpha)$。

这一分析的意义何在呢?若散列表的槽位数正比于其中的元素个数,有 $n=O(m)$,于是,$\alpha=n/m=O(m)/m=O(1)$。因此,查找时间平均为常数。由于插入操作首先需要调用 CHAINED-HASH-SEARCH 确认元素 x 的关键值未曾出现在表中,然后用 $O(1)$时间将 x 插入到链表 $T[h(key[x])]$中,所以期望的时间是 $O(1)$。类似地,删除操作对双向拉链表平均情形时间也是 $O(1)$,所以所有的字典操作可以在 $O(1)$的平均时间内得到支持。

2.5.3 散列表的程序实现

1. 数据类型定义

在 C 语言中,把散列表定义成如下结构体。

```
1 typedef struct {                                      /* hash 表结构 */
2       LinkedList **table;                             /* 槽位数组 */
3       size_t m;                                       /* 槽位数 */
4       size_t n;                                       /* 元素个数 */
5 } Hash_Table;
6 HashTable * createTable(int m);                       /* 创建具有 m 个槽位的 hash 表 */
7 void clrTable(Hash_Table * t);
8 int tbIsEmpty(Hash_Table * t);                        /* 检测表空 */
9 int inHashTable(Hash_Table * t,unsigned key);         /* 向表 t 中插入元素 key */
10 int hashInsert(Hash_Table * t,unsigned key);         /* 在表 t 中查找关键值 key */
11 int hashDelete(Hash_Table * t,unsigned key);         /* 在表 t 中删除元素 key */
```

程序 2-30 定义散列表数据结构及声明操作函数的头文件 hstb. h

对程序 2-30 的说明如下。

(1) 第 1～5 行定义的是散列表类型 HashTable。它有 3 个数据属性:槽位数组 table,该数组的元素是存储表元素的链表;槽位数 m 和表内元素个数 n,它们都是整型的。

(2) 第 6 行、第 7 行声明的函数 createTable 和 clrTable 用来创建散列表和清理散列表的存储空间。第 8 行声明的函数 tbIsEmpty 用来检测散列表是否为空。

(3) 第 9 行～第 11 行声明的函数 inHashTable、hashInsert 和 hashDelete 将分别实现算法 2-24 中的 CHAINED-HASH-SEARCH、CHAINED-HASH-INSERT 和 CHAINED-HASH-

DELETE 过程等字典操作。

2. hash 表的常规维护

hash 表的常规维护操作包括创建空表、清理不再使用的表空间以及检测是否表空。这些操作的 C 函数定义如下。

```
1 HashTable * createTable(int m) {
2     int i;
3     HashTable * tab=(Hash_Table *) malloc(sizeof (Hash_Table));   /* 分配空间 */
4     assert(tab);
5     tab->m=m;                                                      /* 设置槽位数 */
6     tab->table=(LinkedList**) malloc(m * sizeof (LinkedList *));  /* 分配槽位 */
7     assert(tab->table);
8     tab->n=0;                                                     /* 元素个数初始化 */
9     for (i=0; i<m; i++)
10        tab->table[i]=createList(sizeof(unsigned),intGreater);    /* 创建槽位链表 */
11    return tab;
12 }
13 void clrTable(Hash_Table * t) {
14    int i;
15    for (i=0; i<t->m; i++)
16        clrList(t->table[i],NULL);                                /* 清理槽位链表 */
17    free(t->table);                                               /* 释放槽位 */
18 }
19 int tbIsEmpty(Hash_Table * t) {
20    return t->n == 0;
21 }
```

程序 2-31　hash 表的常规维护函数

对程序 2-31 的说明如下。

(1) 第 1～12 行定义的函数 createTable 创建一个具有参数 m 个槽位的散列表(第 6 行)。第 9～10 行的 **for** 循环为每一个槽位创建一个链表存储整型数据的链表,因为 hash 表中元素的关键值是非负整数。

(2) 第 13～18 行定义的函数 clrTable 对参数 t 指引的散列表清理存储空间。其中,第 16 行调用链表清理函数清理每个槽位链表,由于连表中存储的结点数据是普通的非负整数类型数据,无须对链表结点的 key 域做特殊处理,故传递给函数 clrList 的第 2 个参数为 NULL。

(3) 第 19～21 行定义的函数 tbIsEmpty 检测由参数 t 指引的散列表是否为空,它以 t 的 n 属性是否为 0 作为判断依据(第 20 行)。

3. hash 表的字典操作

集合的字典操作包括查找特定值元素、添加元素和删除元素。

```
1 int inHashTable(Hash_Table * t,int key) {              /* 检测 key 是否在表中 */
2     int index;
3     ListNode * node;
4     if (tbIsEmpty(t))
5         return 0;
6     index= key % (t->m);                                /* 计算 key 的 hash 函数值 */
7     node=listSearch(t->table[index],&key);              /* 到指定槽位中去查找 */
8     if (node!=t->table[index]->nil)                     /* 若找到 */
9         return 1;
10    return 0;
11 }
12 int hashInsert(Hash_Table * t,unsigned key) {
13    if (!inHashTable(t,key)) {                          /* 若表中无 key */
14        int index=key%t->m;                             /* 计算 key 的 hash 函数值 */
15        listPushFront(t->table[index],&key);            /* 在指定槽位连表中插入 */
16        t->n++;
17        return 1;
18    }
19    return 0;                                           /* 表中已有 key */
20 }
21 int hashDelete(Hash_Table * t,unsigned key) {
22    int index;
23    ListNode * node;
24    if (tbIsEmpty(t))
25        return 0;
26    index=key % t->m;                                   /* 计算 key 的 hash 函数值 */
27    node=listSearch(t->table[index],&key);              /* 在指定槽位连表中查找 */
28    if (node!=t->table[index]->nil) {                   /* 找到 */
29        listDelete(t->table[index],node);
30        t->n--;
31        free(node);
32        return 1;
33    }
34    return 0;                                           /* 表中无 key */
35 }
```

程序 2-32　定义散列表字典操作的 C 函数

对程序 2-32 说明如下。

(1) 第 1～11 行定义的函数 inHashTable 实现算法 2-24 中的 CHAINED-HASH-SEARCH 过程。该函数在参数 t 指定的散列表中查找参数 key 确定的关键值。若表中存在这个关键值,返回 1,否则返回 0。第 6 行计算 key 的 hash 函数值 index,第 7 行调用程序 2-2 定义的函数 listSearch 在槽位链表 table[index]中查找元素。

(2) 第 12～20 行定义的函数 hashInsert 实现算法 2-24 中的 CHAINED-HASH-INSERT。该函数在参数 t 指定的散列表中插入关键值为参数 key 的元素。插入成功,返回 1,否则返

回 0。第 13 行调用 inHashTable 函数在 t 中查找 key，若表中不存在这样的元素，第 14 行计算 key 的 hash 函数值 index，第 15 行调用程序 2-4 中的函数 listPushFront 在槽位链表 table[index]的前端插入关键值为 key 的结点。

(3) 第 21～35 行定义的函数 hashDelete 实现算法 2-24 中的 CHAINED-HASH-DELETE 过程，在参数 t 指定的散列表中删除关键值为参数 key 的元素，删除成功返回 1，否则返回 0。第 26 行计算 key 的 hash 函数值 index，第 27 行直接调用程序 2-4 中的函数 listSearch 在槽位链表 table[index]中查找 key，第 28 行若检测到存在这样的结点，则在第 29 行调用函数 listDelete(也定义在程序 2-4 中)将该结点删除，并在第 31 行中释放其空间。

为便于代码重用，将程序 2-30 存储为文件夹的 datastructure 中的头文件 hstb. h，将程序 2-31 和程序 2-32 的代码存储为同一个文件内的源文件 hstb. c。

2.5.4　散列表的应用

散列表作为动态集合，常用于需要进行快速查找的应用中。用此数据结构来解决下列问题。

Crazy Search

Description

Many people like to solve hard puzzles some of which may lead them to madness. One such puzzle could be finding a hidden prime number in a given text. Such number could be the number of different substrings of a given size that exist in the text. As you soon will discover, you really need the help of a computer and a good algorithm to solve such a puzzle.

Your task is to write a program that given the size, N, of the substring, the number of different characters that may occur in the text, NC, and the text itself, determines the number of different substrings of size N that appear in the text.

As an example, consider $N=3$, $NC=4$ and the text "daababac". The different substrings of size 3 that can be found in this text are: "daa", "aab", "aba", "bab", "bac". Therefore, the answer should be 5.

Input

The first line of input consists of two numbers, N and NC, separated by exactly one space. This is followed by the text where the search takes place. You may assume that the maximum number of substrings formed by the possible set of characters does not exceed 16 Millions.

Output

The program should output just an integer corresponding to the number of different substrings of size N found in the given text.

Sample Input

```
3 4
```

daababac

Sample Output

5

1. 问题与算法的描述

该问题可形式化地描述如下。

输入：字符串 *text*，*text* 中包含的不同字符个数 *NC*，子串长度 *N*。

输出：*text* 中长度为 *N* 的不同子串个数。

解决此问题的思想非常简单，设 *S* 为一动态集合，初始化为空。从 *text* 首起，逐一提取长度为 *N* 的字串 *substr*，若 $substr \notin S$，则将 *substr* 添加到 *S* 中。最后返回 *S* 中的子串个数即为所得。动态集合 *S* 可以用链表、二叉搜索树等各种数据结构加以表示。然而，本题要求存储在集合 *S* 中的子串要求两两不等，这恰好符合散列表中元素关键值两两不等的特性。重要的是如何将串转换成散列表中元素的整数关键值。转换的思路是，建立一个 *NC* 进制整数系统，*text* 中的每一个字符对应该系统中的一个数字，*text* 中的每一个长度为 *N* 的子串，唯一地对应该整数系统的一个 *N* 位数，将此数作为该子串加入到散列表中的关键值。具体算法伪代码描述如下。

```
CRAZY-SEARCH(text, N, NC)
1 len←length[text]
2 为整型数组 b 分配 256 个元素空间并将每个元素初始化为 0
3 for i←0 to len－1                     ▷确定组成 text 的不同字符
4   do b[text[i]]←1
5 v←0
6 for i←0 to 255
7   do if b[i]≠0                        ▷对 text 中的每个不同字符确定 NC 进制编码值
8     then b[i]←v
9       v←v+1
10 h←∅                                  ▷创建空的 hash 表 h
11 for i←0 to len－N－1                 ▷对 text 的每一个长度为 N 的子串
12   do key←0
13     for j←0 to N－1                  ▷计算该子串对应的 NC 进制值
14       do key←key * NC+b[text[i+j]]
15     HASH-INSERT(h, key)              ▷hash 表仅能插入关键值不重复的元素
16 return size[h]
```

算法 2-27 解决 Crazy Search 问题的 CRAZY-SEARCH 过程

算法过程中设置一个具有 256 个元素的数组 b，用来计算 text 中的 *NC* 个字符对应 *NC* 进制整数系统的数字值。第 3 行和第 4 行的 **for** 循环通过扫描 text 标记出其中 *NC* 个字符。第 6～9 行的 **for** 循环为此 *NC* 个字符计算对应的数字值。第 13 行和第 14 行的 **for** 循环计算一个子串对应的关键值。

2. 程序实现

利用程序 2-30、程序 2-21 及程序 2-23 开发的散列表 Hash_Table，我们将解决 Crazy

Search 问题的算法 2-27 的 CRAZY-SEARCH 过程实现为如下函数。

```
1 int cracySearch(char * text,int n,int nc){
2     int i,j,len=strlen(text),v=0,result;
3     int b[256]={0};
4     Hash_Table * h;
5     i=0;
6     while( * (text+i)!='\0'){                /* 确定组成串 text 的字符 */
7         b[ * (text+i)]=1;
8         i++;
9     }
10    for(i=0;i<256;i++)                       /* 对组成串 text 的各字符确定编码值 */
11        if(b[i])
12            b[i]=v++;
13    h=createTable((int)((len-n) * 0.75));    /* 装载因子 α 设为 0.75 */
14    for(i=0;i<=len-n;i++){
15        int key=0;
16        for(j=0;j<n;j++)                     /* 计算子串的关键值 */
17            key=key * nc+b[ * (text+i+j)];
18        hashInsert(h,key);
19    }
20    result=h->size;                          /* 不同的子串数 */
21    clrTable(h,NULL);
22    free(h);
23    return result;
24 }
```

程序 2-33　实现算法 2-25 CRAZY-SEARCH 过程的 C 函数

函数代码与算法伪代码十分相似，需要注意的是第 13 行调用函数 createTable 创建散列表 h 时，参数(len－n) * 0.75 表示用装载因子 α＝0.75 决定散列表的槽位数。其中 len 是串 text 的长度，n 是子串长度，len-n 恰为 text 中所有长度为 n 的子串数，也就是本问题中可能加入 hash 表的最大元素个数。调用该函数解决 Crazy Search 问题的程序存放在文件夹 chap02/Crazy Search 的源文件 CrazySearch.c 中。

第 3 章　基本算法设计策略

人们在解决复杂的问题时，思维方向通常有两个：集零为整和化整为零。集零为整的思路就是以累加或扩张的方式由点连线，由线及面，从部分的解扩展成完整的解。化整为零的思路则是相反，将一个复杂的问题分解成较小的问题，继续分解，直至能直接解得为止。这样的思维方法，用在算法设计上就形成了两种基本的算法设计策略：渐增策略和分治策略。本章通过一些简单而又经典的例子说明这些策略的应用。

3.1　渐增型算法

从讨论一类最简单的算法设计策略——渐增型算法开始。**渐增型算法**指的是算法使问题的解从较小的部分渐渐扩张，最终成长为问题的完整解。运用渐增型策略的算法有一个共同的特征：构成算法的主体是一个循环结构，它逐步将部分解扩展成一个完整解。该循环将遵循一个始终不变的原则：每次重复之初，总维持着问题的一个部分解。人们将此特征称为算法的**循环不变量**。利用循环不变量来证明渐增型算法的正确性是软件正确性证明的一个很好的方法。具体的做法是，先用数学归纳法证明循环不变量的正确性，然后利用循环不变量说明主循环结束时将得到关于问题的一个完整解。本节介绍两个典型的渐增型算法，讨论它们的正确性，指出它们的效率，并用 C 语言加以实现。

3.1.1　有序序列的合并问题

1. 问题的描述

将两个有序序列**合并**为一个有序序列问题的形式化表示如下。

输入：序列 $A[p..r]$。其中，子序列 $A[p..q]$和 $A[q+1..r]$是有序的。

输出：$A[p..r]$所有元素的重排，使之有序。

可以用一个渐增型算法解决此问题。首先，把 $A[p..q]$和 $A[q+1..r]$分别复制到序列 $L[1..n_1]$和 $R[1..n_2]$中，其中 $n_1=q-p+1$，$n_2=r-q$。然后，维护 3 个变量 i、j 和 k，i、j 初始化为 1，k 初始化 p。比较 $L[i]$与 $R[j]$，将较小者复制到 $A[k]$，调整 i、j 和 k 使得它们各自指向 L、R 和 A 的合适的位置：若 $L[i]\leqslant R[j]$，则 i 增加 1，否则 j 增加 1，无论如何 k 都要增加 1。循环往复，直至 L 和 R 之一被扫描完。然后，将另一个序列中尚存的元素复制到 $A[k..r]$。在此过程中，$A[p..k-1]$中的元素是 $A[p..r]$中的最小的 $k-p$ 个元素，并且已排好序。随着 k 的增长，$A[p..k]$渐增为一个有序序列。图 3-1 展示了一个例子。

图 3-1 所示为当子数组 $A[9..16]$含有序列＜2，4，5，7，1，2，3，6＞时，$A[9..12]$和 $A[13..16]$为两个有序的子序列。使用两个辅助序列 L 和 R 进行合并操作。在复制后，序

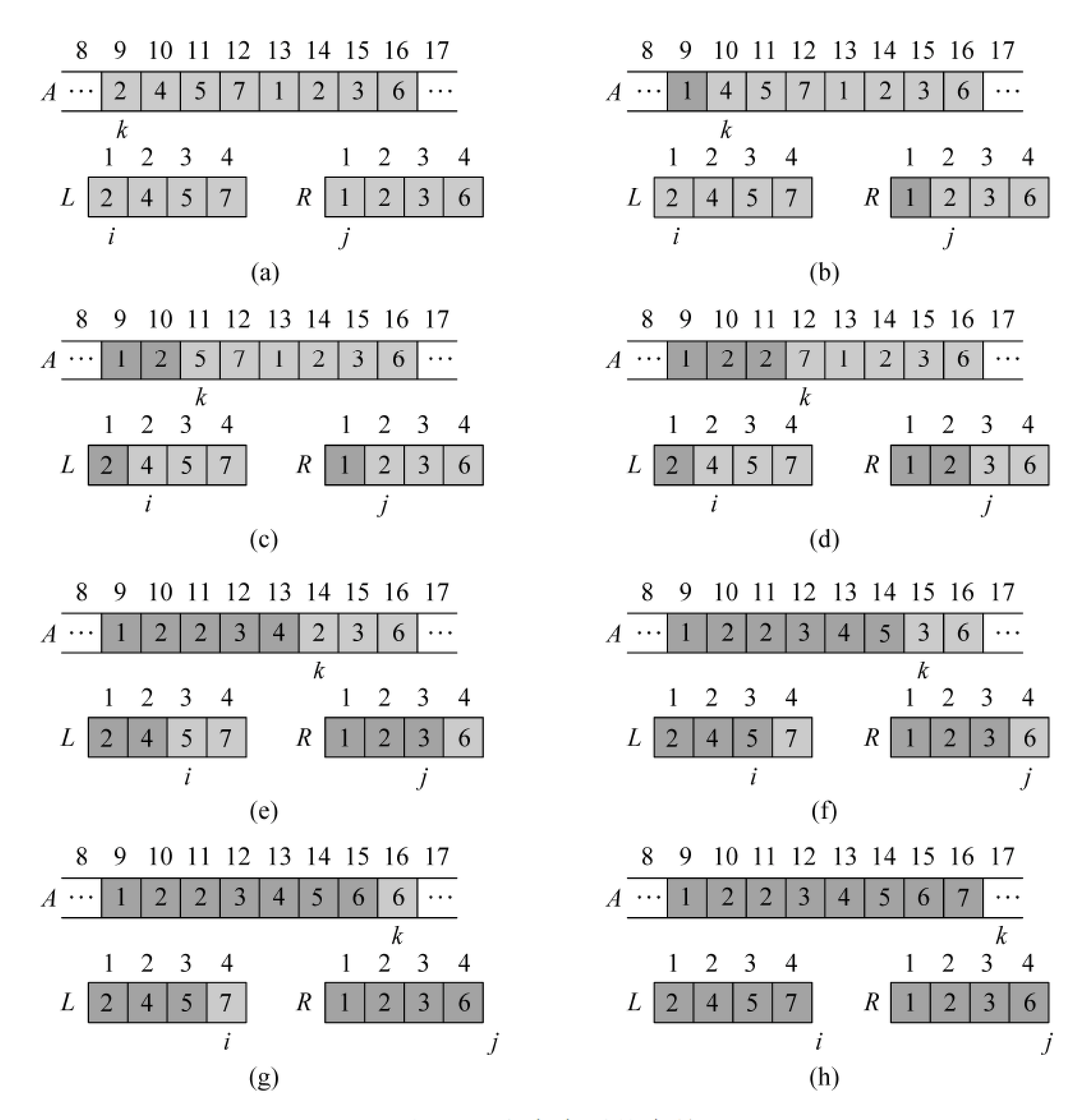


图 3-1　有序序列的合并

列 L 包含<2,4,5,7>，序列 R 包含<1,2,3,6>。A 中深灰色的位置包含最终值，L 和 R 中浅灰色位置包含的值尚未复制回 A 中。A 的浅灰色位置是将被覆盖的，而 L 和 R 中深灰色位置包含的是已经被复制回 A 的值。图 3-1(a)～图 3-1(g)是算法 3-1 中第 12～17 行的 **while** 循环每次重复之初的 A、L、R 及其下标 k、i、j。图 3-1(h)是执行了算法 3-1 中第 18～21 行将 L 中剩余的元素复制到 $A[k..r]$中。此时，子序列 $A[9..16]$已经排好序。

2. 算法的伪代码描述

很容易将上述的算法思想描述为下列的伪代码过程。

```
MERGE(A,p,q,r)
1 n1←q-p+1
2 n2←r-q
3 创建数组 L[1..n1]和 R[1..n2]
4 for i←1 to n1
5     do L[i]←A[p+i-1]
6 for j←1 to n2
7    do R[j]←A[q+j]
8 i←1
```

```
9 j←1
10 k←p
11 while i≤n1 and j≤n2
12    do if L[i]≤R[j]
13          then A[k]←L[i]
14                i←i+1
15          else A[k]←R[j]
16                j←j+1
17       k←k+1
18 if i<n1
19    then 将 L[i..n1]复制到 A[k..r]
20 if j<n2
21    then 将 R[j.. n2]复制到 A[k..r]
```

算法 3-1 解决有序子序列合并问题的 MERGE 算法

3. 算法的正确性

MERGE(A,p,q,r)是把存储在 $A[p..q]$和 $A[q+1..r]$这两部分中的有序子序列合并到 $A[p..r]$并使其有序。随着第 12～17 行的 **for** 循环的重复,部分解 $A[p..k]$逐渐扩充为全部解 $A[p..r]$。由此可总结出如下的循环不变量。

在第 12～17 行的 **while** 循环的每次重复之初,子数组 $A[p..k-1]$包含 $L[1..n_1]$和 $R[1..n_2]$中的 $k-p$ 个最小的元素,并排好序。此外,$L[i]$和 $R[j]$分别是各自数组中尚未复制回数组 A 的元素中的最小者。

利用此循环不变量来证明合并算法 MERGE 是正确的。

首先,对变量 k 做数学归纳。

(1) 初始时,$k=p$。此时,$A[p..k-1]=\varnothing$。此外,$L[1]$、$R[1]$分别是各自数组中尚未复制回数组 A 的元素中的最小者。所以,循环不变量自然为真。

(2) 设 $p<k<r$,且重复之初循环不变量为真,即 $A[p..k-1]$包含 $L[1..n_1]$和 $R[1..n_2]$中的 $k-p$ 个最小的元素,并排好序。此外,$L[i]$和 $R[j]$分别是各自数组中尚未复制回数组 A 的元素中的最小者。在此假设下,本次重复中,第 13～16 行将把 $L[i]$和 $R[j]$两者的较小者复制到 $A[k]$处,根据 L、R 的有序性知,此时 $A[p..k]$包含 $L[1..n_1]$和 $R[1..n_2]$中的 $k-p+1$ 个最小的元素,并排好序。若 $L[i]$较小,则 i 增加 1,否则 j 增加 1。此时仍有 $L[i]$和 $R[j]$分别是各自数组中尚未复制回数组 A 的元素中的最小者。这一状态将持续到下一次重复之初。那时,k 增值 1,则此状态应描述为:$A[p..k-1]$包含 $L[1..n_1]$和 $R[1..n_2]$中的 $k-p$ 个最小的元素,并排好序。此外,$L[i]$和 $R[j]$分别是各自数组中尚未复制回数组 A 的元素中的最小者。这恰为循环不变量!

这样就证明了上述循环不变量是正确的。于是,第 12～17 行的 **while** 循环终止时,$i>n_1$ 或 $j>n_2$。根据循环不变量知,$A[p..k-1]$包含 $L[1..n_1]$和 $R[1..n_2]$中的 $k-p$ 个最小的元素,并排好序。第 18～21 行将尚存于 L 或 R 中的 $r-k+1$ 个元素(是有序的)复制到 $A[k..r]$中,使得 $A[p..r]$为有序,这正是要求的结果。

4. 算法的运行时间

假定 $A[p..r]$含有 $n(n=r-p+1)$个元素。第 4 行～第 7 行的 **for** 循环均重复 n 次。此外，第 12～17 行的 **while** 循环重复 $k-p$ 次，第 18～21 行中的 **for** 循环将重复 $r-p+1$次，两者一共耗时 $\Theta(n)$。所以算法的最坏情形时间为 $\Theta(n)$。

5. 程序实现

这个过程是一个有效的排序算法——归并排序中的关键过程，所以，要来实现它。

1）**数组版本**

首先，利用 **void** * 指针的方法来实现一个能对任意类型数组进行合并的通用的C 函数。为便于代码重用，在头文件 merge.h 中声明函数：

```
void merge(void *a,int size,int p,int q,int r,int(*comp)(void *,void *));
```

该函数对存储在指针 a 指向的内存中每个元素的存储宽度为 size 的数组中有序子数组 a[p..q]和 a[q+1..r]进行合并。数组中的元素大小比较规则由参数 comp 确定。该函数的定义如下。

```
void merge(void *a,int size,int p,int q,int r,int(*comp)(void *,void *)){
    int i,j,k,n1=q-p+1,n2=r-q;
    void *L=(void*)malloc(n1*size);
    void *R=(void*)malloc(n2*size);
    memcpy(L,(char*)a+p*size,n1*size);          /*将 a[p..q]复制到 L[1..n1]*/
    memcpy(R,(char*)a+(q+1)*size,n2*size);      /*将 a[q+1..r]复制到 R[1..n2]*/
    i=j=0;
    k=p;
    while(i<n1&&j<n2)
        if(comp(L+i*size,R+j*size)<0)                  /*L[i]<R[j]*/
            memcpy(a+(k++)*size,L+(i++)*size,size);    /*a[k]←L[i]*/
        else
            memcpy(a+(k++)*size,R+(j++)*size,size);    /*a[k]←R[j]*/
    if(i<n1)
        memcpy(a+k*size,L+i*size,(n1-i)*size);     /*将 L[i..n1]复制到 a[k..r]*/
    if(j<n2)
        memcpy(a+k*size,R+j*size,(n2-j)*size);     /*将 R[j..n2]复制到 a[k..r]*/
    free(L);L=NULL;
    free(R);R=NULL;
}
```

程序 3-1　实现算法 3-1 的 C 函数 merge

对程序 3-1 的说明如下。

（1）函数 merge 有 6 个参数，其中 a、p、q、r 与 MERGE 过程的参数 A、p、q、r 对应。由于 a 是 **void** * 类型的，所以本函数可以对各种类型的数组做合并操作。参数 size 的意义是指定存储在数组 a 中的元素的实际长度，参数 comp 的意义是确定数组 a 中元素大小比较的

规则。

(2) 第 5 行和第 6 行调用库函数 memcpy 实现 MERGE 过程中的第 4 行和第 5 行两行子数组复制的操作。第 9～13 行的 **while** 循环对应 MERGE 过程中第 9～15 行的 **while** 循环。注意,对**void** * 指针指向内存单元的赋值操作由函数 memcpy 完成。第 14～15 行和第 16 行和第 17 行分别对应 MERGE 过程中第 16～19 行的剩余子数组复制操作。

(3) 第 18 行和第 19 行释放指针 L、R 指示的内存空间,以防内存泄漏。

为便于重用,把 merge 函数写在源文件 merge. c 中,并连同头文件 merge. h 存放在目录 Utility 中。

2) **链表版本**

事实上,序列在计算机中既可以表示成数组,也可以表示成链表。有序序列的合并问题,也可能发生于用链表表示的序列上。利用第 2 章讨论过的通用双向链表 LinkedList,可以在 C 语言中将 MERGE 过程实现为如下链表版本。

```
void listMerge(LinkedList *A,ListNode *p,ListNode *q,ListNode *r){
    LinkedList *L=createList(A->eleSize,A->comp),
               *R=createList(A->eleSize,A->comp);
    ListNode *i,*j,*k;
    assert(L&&R);
    i=p;
    do{                                                /*将 a[p..q]复制到 L*/
        listPushBack(L,i->key);
        i=i->next;
    }while(i!=q->next);
    j=q->next;
    do{                                                /*将 a[q+1..r]复制到 R*/
      listPushBack(R,j->key);
      j=j->next;
    }while(j!=r->next);
    i=L->nil->next,j=R->nil->next,k=p;
    while(i!=L->nil&&j!=R->nil){
      if(A->comp((char*)i->key,(char*)j->key)<0){      /*L[i]<R[j]*/
        memcpy((char*)k->key,(char*)i->key,A->eleSize);/*A[k]←L[i]*/
        i=i->next;                                     /*i←i+1*/
      }else{
        memcpy((char*)k->key,(char*)j->key,A->eleSize);/*A[k]←R[j]*/
        j=j->next;                                     /*j←j+1*/
      }
      k=k->next;                                       /*k←k+1*/
    }
    if(i!=L->nil)
      while(i!=L->nil){                                /*将 L[i..n1]复制到 A[k..r]*/
        memcpy((char*)k->key,(char*)i->key,A->eleSize);
        i=i->next,k=k->next;                           /* i←i+1,k←k+1*/
      }
```

```
31  if(j!=R->nil)
32    while(j!=R->nil){                              /* 将 R[j..n2]复制到 A[k..r] */
33      memcpy((char *)k->key,(char *)j->key,A->eleSize);
34      j=j->next,k=k->next;                         /* j←j+1,k←k+1 */
35    }
36  clrList(L,NULL),free(L);
37  clrList(R,NULL),free(R);
38}
```

程序 3-2　实现算法 3-1 的 C 函数链表版本 listMerge

对程序 3-2 的说明如下。

(1) 函数 listMerge 有 4 个参数。参数 A 表示链表 A[p..r],参数 p、q、r 分别是指向序列 A 的首元素、中点元素和尾元素的指针。

(2) 与程序 3-1 中函数 merge 相比,L、R 分别定义成链表类型而非数组。同时,变量 i、j、k 也定义成了指向链表结点的指针类型而非表示数组下标的整型。

(3) 将程序 3-1 中所有对数组元素的访问替换成对链表结点的访问,下标的自增换成取下一个结点的操作,就形成了程序 3-2 中相应的代码。注意,第 36 行和第 37 行调用函数 clrList 清理链表 L、R 的空间,然后释放指针 L、R。

为便于重用,将函数 listMerge 的定义代码存储为 utility 文件夹内的源文件 listmerge.c,并将该函数的原型声明

```
void listMerge(LinkedList *A,ListNode *p,ListNode *q,ListNode *r);
```

存储在头文件 listmerge.h 中。

3.1.2　序列的划分问题

1. 问题的描述

在序列的划分问题中,操作的对象是一个表示成序列的全序集。操作的目标是将该序列分成前后两部分,使得前一部分元素的值小于后一部分中元素的值。形式化描述如下。

输入:表示成序列的全序集 $A[p..r]$。

输出:下标 $q(p\leqslant q\leqslant r)$,原序列 $A[p..r]$的一个重排:使得 $A[p..q]$中的元素值不超过 $A[q]$,而 $A[q+1..r]$中的元素值均大于 $A[q]$。

解决此问题的算法对序列 $A[p..r]$进行原地重组。算法选择元素 $x=A[r]$作为**基准**元素,以它为分界点对序列 $A[p..r]$进行划分,目标是将其分成前后两段 $A[p..q]$和 $A[q+1..r]$,使得 $A[p..q]$中的元素值不超过 x,而 $A[q+1..r]$中的元素值大于 x。算法维护两个下标值 i 和 j,初始时分别为 $p-1$ 和 p。让 j 在$[p..r)$中扫描,若 $A[j]\leqslant A[r]$,则将 $A[j]$与 $A[i+1]$交换,然后 i 增加 1。这样,在此过程中,$A[p..i]$中的元素均不超过 $A[r]$,而 $A[i+1..j-1]$中的元素大于 $A[r]$。随着 j 的增加,$A[p..i]$和 $A[i+1..j-1]$也随之增长,最终 j 达到 $r-1$,并将 $A[i+1]$与 $A[r]$交换,返回 $i+1$ 即为所求的 q。例如,对图 3-2 中所示的序列进行的操作。

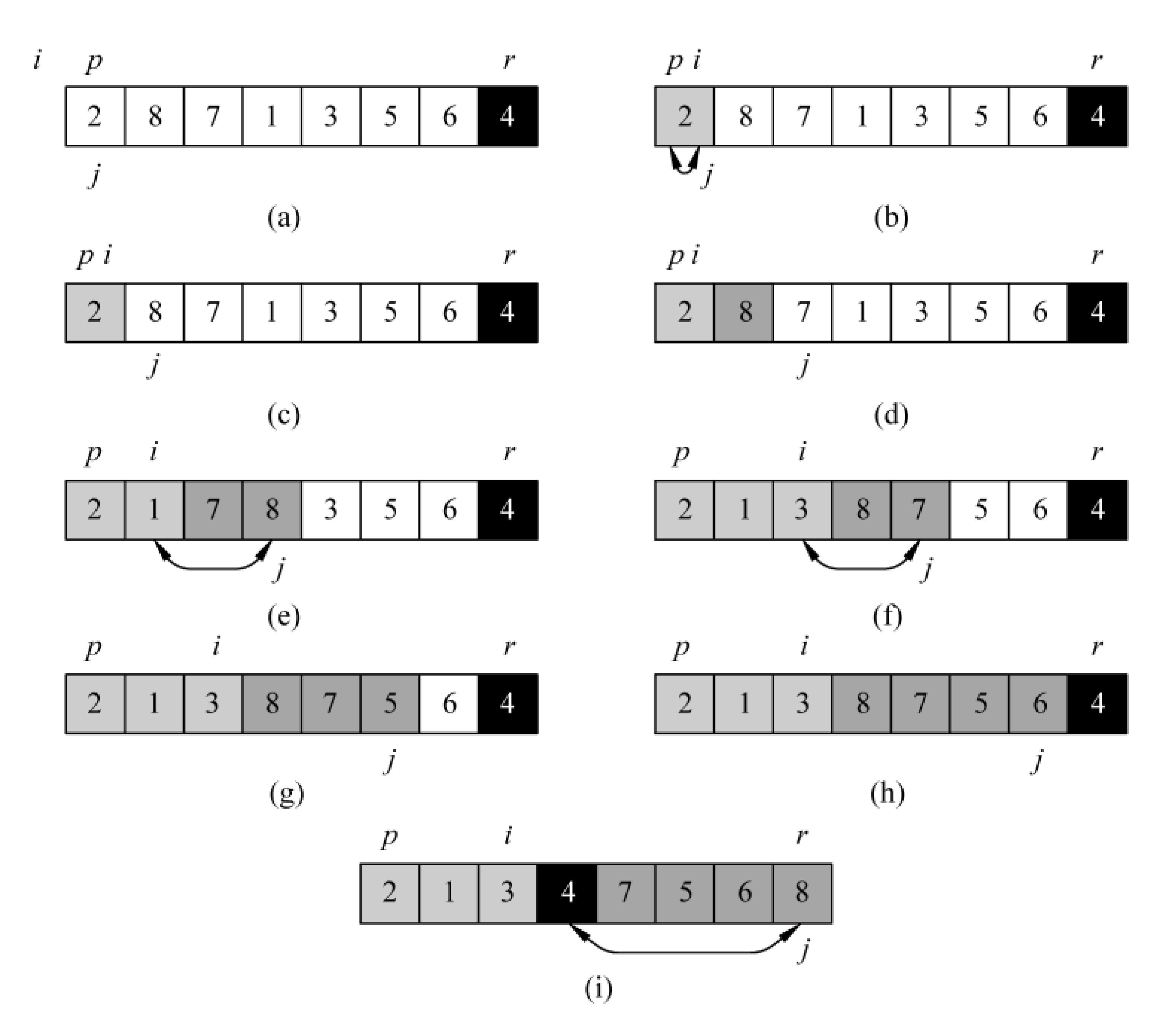


图 3-2　序列划分操作

图 3-2 所示为对样本序列＜2,8,7,1,3,5,6,4＞的划分操作。浅阴影序列元素都在第一部分 $A[p..i]$中,其值都不大于 x。深阴影元素都在第二部分 $A[i+1..j-1]$中,其值都大于 x。无阴影元素是尚未进入上述两个部分的元素,黑色元素是基准元素。图 3-2(a)为初始的序列和变量设置。没有任何元素进入上述两部分。图 3-2(b)～图 3-2(h)表示算法 3-2 中第3～6 行的 **for** 循环的每一次重复。黑色双向箭头表示第 6 行的元素交换操作。图 3-2(i)表示上述循环终止后第 7 行执行的元素交换操作。

2. 算法的伪代码描述

```
PARTITION(A,p,r)
1 x←A[r]
2 i←p−1
3 for j←p to r−1
4      do if A[j]≤x
5            then i←i+1
6                  exchange A[i]↔A[j]
7 exchange A[i+1]↔A[r]
8 return i+1
```

算法 3-2　解决划分问题的 PARTITION 算法

3. 算法的正确性

在此过程的运行中,序列被分成 4 个(可能有些为空)部分: $A[p..i]$、$A[i+1..j-1]$、

$A[j..r-1]$和$A[r]$。$A[p..i]$中的元素值不超过$A[r]$,$A[i+1..j-1]$中的$A[r]$、$A[j..r-1]$中的元素尚未处理过。随着算法中第 3～6 行的 **for** 循环的重复,$A[p..i]$和$A[i+1..j-1]$的范围逐渐扩大,最终使得$A[p..i]$中的元素不超过$A[i+1]$,$A[i+2..r]$中的元素大于$A[i+1]$。返回的$i+1$就是划分的分界点q。在第 3～6 行的 **for** 循环的每次重复之初,每一部分满足一定的性质,这些性质可陈述为一个循环不变量。

在第 3～6 行的 **for** 循环的每次重复之初,对每一个数组下标k:

(1) 若 $p\leqslant k\leqslant i$,则$A[k]\leqslant x$;

(2) 若 $i+1\leqslant k\leqslant j-1$,则$A[k]>x$;

(3) 若 $k=r$,则$A[k]=x$。

用此循环不变量来证明算法 3-2 的正确性。首先,对j做数学归纳。

① $j=p$ 时,$i=p-1$。此时,$[p,i]=\varnothing$。循环不变量中的(1)自然成立。$[i+1,j-1]=\varnothing$,所以循环不变量中的(2)也成立。由第 1 行知,若$k=r$,$A[k]=A[r]=x$,此为循环不变量中的(3)。

② 假定 $p<j<r-1$ 时,循环不变量成立,即本次重复之初条件(1)、(2)、(3)都成立。考察本次重复所执行的操作:如果$A[j]\leqslant x$,则将$A[j]$与$A[i+1]$交换,且i增加 1。这样$A[p..i]$中的元素不超过x,下次重复之初就使得循环不变量的条件(1)成立。如果$A[j]>x$,i不变,也不做其他任何操作。这使得下次重复之初保持本次重复之初的条件(1)成立。总之,无论如何,本次重复结束时的状态,也就是下次重复之初的状态将维持条件(1)的成立。由于本次重复之初$A[i+1..j-1]$中元素大于x。本次重复中,若发生$A[j]$与$A[i+1]$交换($A[j]<x$),则i将增加 1 且此时$A[j]>x$,也就是说无论如何$A[i+1..j]$中元素大于x。本次重复完成下次重复之初,j将增加 1,这意味着$A[i+1..j-1]$中元素大于x,这是循环不变量的条件(2)。至于条件(3),则所有的重复中,$A[r]$没有发生变化,始终为x。

由此可见,该算法的循环不变量是正确的。在第 3～6 行的 **for** 循环终止时,按循环不变量$A[p..i]$中元素不超过x,$A[i+1..r-1]$中元素大于x。第 7 行交换$A[i+1]$和$A[r]$就使得$A[p..i+1]$中元素不超过$A[i+1]$,而$A[i+2..r]$中元素大于$A[i+1]$。这正是要求的划分结果,$i+1$就是分界点元素的下标。

4. 算法的运行时间

假定序列$A[p..r]$中含有n个元素。在此过程中,第 3～6 行的 **for** 循环重复了n次,所以该算法的最坏情形运行时间为$\Theta(n)$。

5. 程序实现

这个过程在很多有趣的算法中都有应用,所以要把它认真地加以实现。

1) 数组版本

由于 C 语言中没有提供交换两个变量的值的库函数,而在程序设计中,经常会遇到这样的操作,例如,在实现 PARTITION 过程时,第 6 行和第 7 行就需要进行这一操作。完成两个变量值交换的 C 源代码文件 swap.c 如下。

```
void swap(void * x,void * y,int size){
    void * temp=(void * )malloc(size);
```

```
3        memcpy(temp,x,size);
4        memcpy(x,y,size);
5        memcpy(y,temp,size);
6        free(temp);
7 }
```

程序 3-3 完成两个变量值交换的 C 源代码文件 swap.c

对程序 3-3 的说明如下。

(1) 函数 swap 有 3 个参数:**void** * 指针型的 x、y 和整型的 size。前两者提供需要交换值的两个变量的地址,第 3 个参数 size 则指出存储于这些地址中的数据的实际长度。之所以要用 **void** * 指针,是希望该函数能适用于任何数据类型。由于交换的结果维持于 x、y 所指向的地址中,所以无须返回值。

(2) 第 5 行声明的局部变量 temp 扮演交换变量值时的中转角色。

(3) 除了用 memcpy 充当赋值操作外,与常做的交换两变量的值的操作逻辑是一样的。

为便于重用,把这段代码存储于 Utility 目录的源文件 swap.c 中,且将该函数的原型声明存储为头文件 swap.h。利用 swap 函数,按照算法 3-2 的伪代码描述,将其实现为如下的 C 函数。

```
1 #include "swap.h"
2 long partition(void *a,int size,int p,int r,int(*comp)(void *,void *)){
3        int i,j;
4        void *x=(void *)malloc(size);
5        memcpy(x,a+r*size,size);                    /* x←a[r] */
6        i=p-1;
7        for(j=p;j<r;j++)
8            if(comp(a+j*size,x)<=0){                /* a[j]≤x */
9                i++;
10               swap(a+i*size,a+j*size,size);       /* a[i]↔a[j] */
11           }
12       free(x);
13       swap(a+(i+1)*size,a+r*size,size);           /* a[i+1]↔a[r] */
14       return i+1;
15 }
```

程序 3-4 实现算法 3-2 的 C 函数 partition 的定义

对程序 3-4 的说明如下。

(1) partition 函数有 5 个参数:其中 a、p、r 对应于 PARTITION 过程的参数 A、p、r。注意,a 的类型是 **void** *,这意味着该函数能对任何类型的数组做划分操作。参数 size 是确定 a 中元素的存储长度,参数 comp 确定 a 中元素大小比较规则。函数将返回分界点元素的下标。

(2) 第 4 行声明 **void** * 型的变量 x 对应于算法中同名的暂存 $A[r]$的值的临时变量。

(3) 只要把伪代码中的赋值操作代之以函数 memcpy 的调用;数组元素 $A[i]$的访问代之以 a+i*size;交换元素操作代之以函数 swap 的调用,则程序代码几乎就是伪代码的

翻版。

为便于重用，把 partition 函数的定义存放在目录 Utility 中的源文件 partition. c 中，并将该函数的原型声明存储为 partition. h。

2）**链表版本**

对于组织成链表的序列，将 PARTTION 过程实现为如下 C 程序。

```
ListNode * listPartition(LinkedList * A,ListNode * p,ListNode * r){
    ListNode * i, * j;
    void * x=(void * )malloc(A->eleSize);
    memcpy((char * )x,(char * )r->key,A->eleSize);          /* x←a[r] */
    i=p->prev;
    for(j=p;j!=r;j=j->next)
        if(A->comp((char * )j->key,(char * )x)<=0){        /* a[j]≤x */
            i=i->next;
            swap((char * )i->key,(char * )j->key,A->eleSize);   /* a[i]↔a[j] */
        }
    free(x);
    swap((char * )i->next->key,(char * )r->key,A->eleSize);     /* a[i+1]↔a[r] */
    return i->next;
}
```

程序 3-5　实现算法 3-2 的 C 函数链表版本的定义

对程序 3-5 的说明如下。

（1）与算法过程一样，函数 listPartition 有 3 个参数。参数 A 表示序列 A[p. . r]，参数 p、r 分别是指向序列的首元素和尾元素的指针。

（2）与程序 3-4 中的函数 partition 相比，变量 i、j 定义成了指向链表结点的指针类型而非表示数组下标的整型。

（3）将程序 3-4 中所有对数组元素的访问替换成对链表结点的访问，下标的自增换成取下一个结点的操作，就形成了程序 3-5 中相应的代码。

为便于重用，将函数 listPartition 的定义代码存储为 utility 文件夹内的源文件 listpartition. c，并将该函数的原型声明存储在头文件 listpartition. h 中。

3.2　分治算法

很多有用的算法是**递归**结构的：为解决一个给定的问题，递归地调用自身一次或多次来解决关系密切的若干个子问题。这样的算法通常遵循**分治**方法：它们将问题分解成若干个与原问题相仿而规模较小的子问题，递归地解决这些子问题，然后把子问题的解合并成原问题的一个解。

分治范式在每一层递归包括 3 个步骤。

（1）**分解**：将问题分解成若干个子问题。

（2）**治理**：递归地解决各子问题。不过，若子问题的规模足够小，就以直接的方式（不

再递归)解决子问题。

(3) **合并**:将子问题的解合并成原问题的一个解。

3.2.1 归并排序算法

1. 排序问题描述

对序列进行排序是很多应用问题中的基本操作。序列的排序问题可形式化地表示为如下。

输入:表示成序列的全序集$<a_1,a_2,\cdots,a_n>$。

输出:输入的一个排列(重排)$<a_1',a_2',\cdots,a_n'>$,满足 $a_1'\leqslant a_2'\leqslant\cdots\leqslant a_n'$。

研究序列排序问题的历史十分悠久,人们用各种各样的策略,设计出了种种解决此问题的算法。相信很多读者都是从解决冒泡排序、插入排序等问题中开始程序设计学习的。本节讨论两个用分治策略设计的排序算法,其中之一就是归并排序算法。

2. 算法的伪代码描述

归并排序是一个典型的分治算法:将序列 $A[p..r]$分成 $A[p..q]$和 $A[q+1..r]$,分别递归地对这两个子序列排序,将得到的排好序的子序列 $A[p..q]$和 $A[q+1..r]$调用在前面讨论过的 MERGE 过程合并成整个有序序列 $A[p..r]$。图 3-3 示例了归并排序的过程。

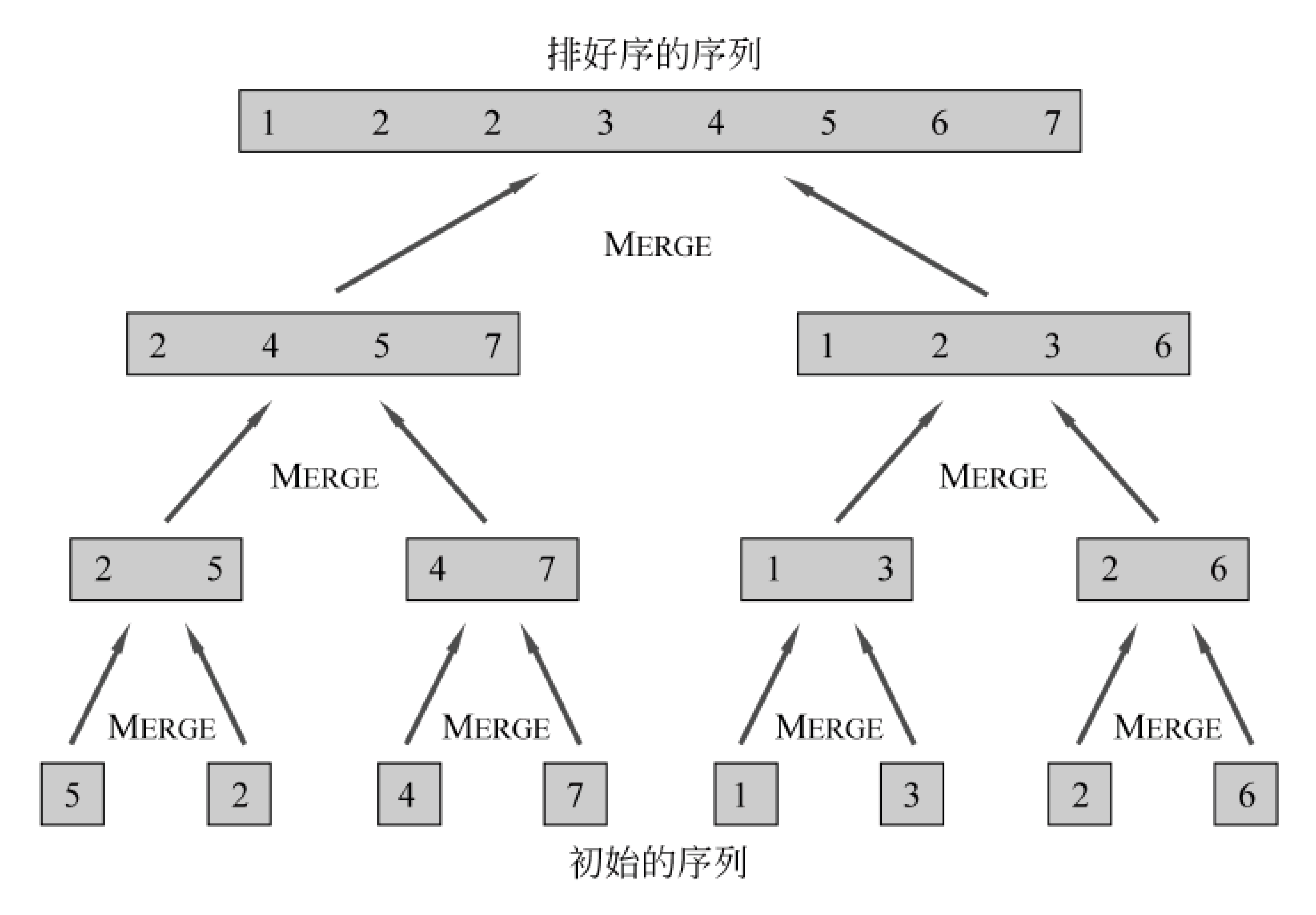


图 3-3 归并排序操作

图 3-3 所示为对数组 $A=<5,2,4,7,1,3,2,6>$进行的归并排序操作。随着算法自底向上进行,排好序的序列长度也在增加。

伪代码描述如下:

```
MERGE-SORT(A,p,r)
1 if p<r
```

```
2   then q←⌊(p+r)/2⌋①
3       MERGE-SORT(A,p,q)
4       MERGE-SORT (A,q+1,r)
5       MERGE(A,p,q,r)
```

算法 3-3　归并排序算法 MERGE-SORT

3. 算法的运行时间

运用分治策略的算法，其运行时间往往表示成一个递归方程。以 MERGE-SORT 过程为例，设 $A[p..r]$中有 n 个元素，设对 $A[p..r]$进行归并排序的运行时间为 $T(n)$。在算法 3-3 中，第 2 行通过一个赋值操作完成对问题的划分，耗时 $\Theta(1)$。第 3 行和第 4 行分别递归调用 MERGE-SORT 过程自身对 $A[p..q]$和 $A[q+1..r]$进行归并排序，由于 $A[p..q]$和 $A[q+1..r]$各含 $n/2$ 个元素，所以这两行耗时 $2T(n/2)$。第 5 行调用 MERGE 过程将有序子序列 $A[p..q]$和 $A[q+1..r]$合并成有序序列 $A[p..r]$。根据本章 3.1.1 节，第 5 行耗时 $\Theta(n)$，于是得到：

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+\Theta(n) & n>1\end{cases} \tag{3-1}$$

解递归方程有多种方法，此处介绍常用的 3 种解法。

1）递归树法

用递归树对分治算法的运行时间进行描述特别直观。在一棵**递归树**中，每一个结点表示一系列递归函数调用中对一个单一子问题的开销。把树的每个层次中的开销相加得到各层次的开销，然后再把各层次开销相加来确定递归树的所有层次上的总开销。

以式(3-1)为例，画出对应的递归树，如图 3-4 所示。

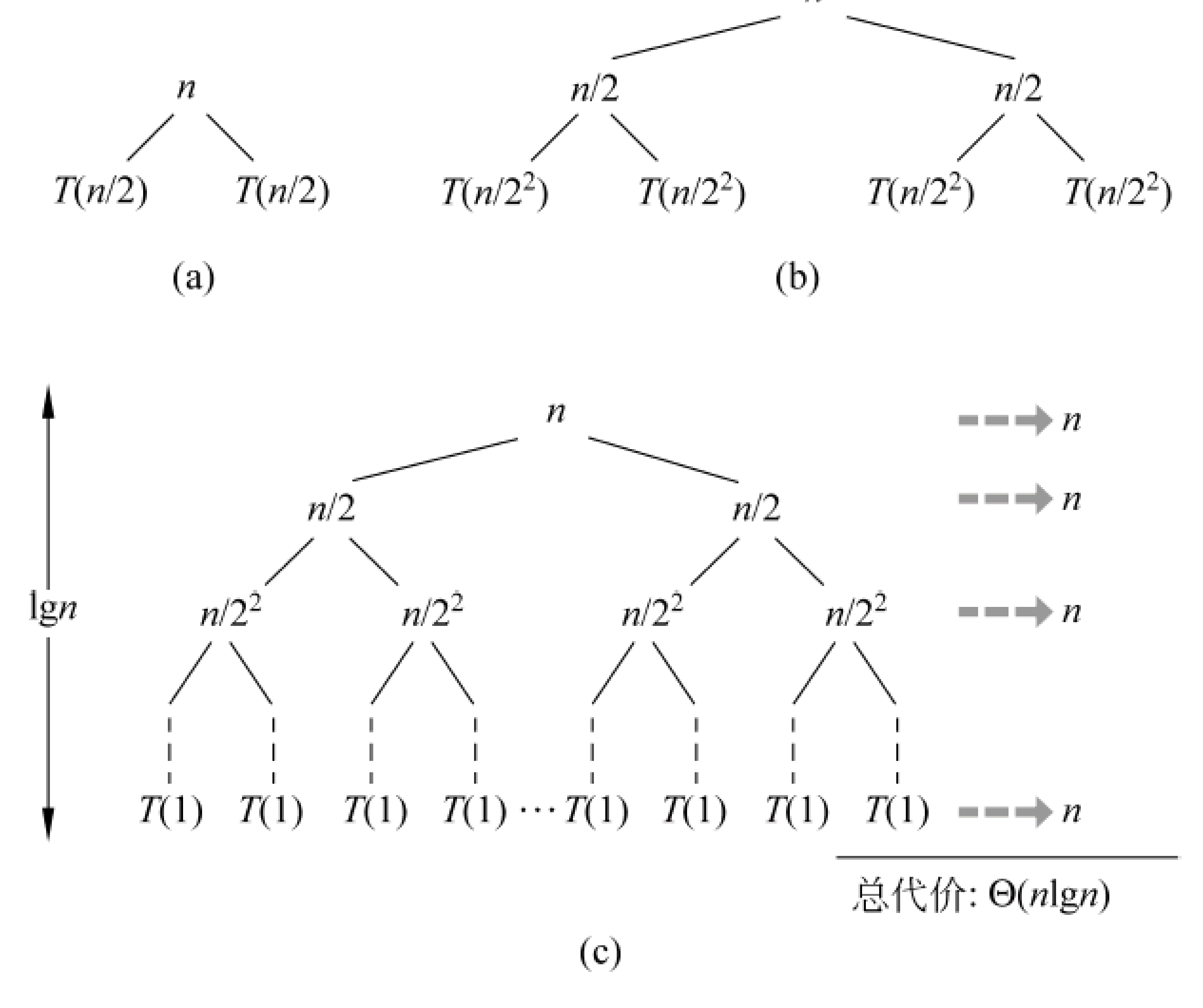


图 3-4　MERGE-SORT 运动时间的递归树法

① $\lfloor x\rfloor$表示不超过 x 的最大整数。例如，$x=3.54$，$\lfloor x\rfloor=3$；而 $x=-3.54$，$\lfloor x\rfloor=-4$。$\lceil y\rceil=3$ 表示不小于 y 的最小整数。例如，$y=3.54$，$\lceil y\rceil=4$；而 $y=-3.54$，$\lceil y\rceil=-3$。

图 3-4 所示为递归式 $T(n)=2T(n/2)+cn^2$ 的递归树的创建。图 3-4(a)部分展示了 $T(n)$,它在图 3-4(b)和图 3-4(c)部分中逐渐展开,形成递归树。图 3-4(c)部分中完全展开的树的高度为 $\lg n$(有 $\lg n+1$ 层)。

由图 3-4 可知,式(3-1)的解为 $T(n)=\Theta(n\lg n)$。也就是说,归并排序算法最坏情形运行时间为 $\Theta(n\lg n)$。

2) **迭代法**

对递归方程,可以根据其定义,逐层迭代直至递归头。仍以式(3-1)为例:

$$\begin{aligned} T(n) &= 2\ T(n/2)+n \\ &= 2(2T(n/2^2)+n/2)+n = 2^2T(n/2^2)+2n \\ &= 2^2(2T(n/2^3)+n/2^2)+2n = 2^3T(n/2^3)+3n \\ &\ \ \vdots \\ &= 2^{\lg n}+n\lg n \\ &= \Theta(n\lg n) \end{aligned}$$

与上述递归树法得到的结果是一样的。

3) **定理法**

对形如式(3-1)这样的递归方程,可以利用以下的定理解它。

定理 3-1 设 a、b、c 是非负常数,n 是 c 的幂。递归方程

$$T(n)=\begin{cases} b & n=1 \\ aT(n/c)+O(n) & n>1 \end{cases}$$

的解为

$$T(n)=\begin{cases} O(n) & a<c \\ O(n\lg n) & a=c \\ O(n^{\log_c a}) & a>c \end{cases}$$

根据此定理,由于 $a=2$,$c=2$,$T(n)=\begin{cases} 1 & n=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$,该算法的解为 $\Theta(n\lg n)$。

4. 程序实现

1) **数组版本**

利用 3.1.1 节开发的合并操作的数组版本的 merge 函数,把算法 3-3 实现为以下 C 函数。

```
#include "merge.h"
void mergeSort(void *a,int size,int p,int r,int(*comp)(void *,void *)){
    if(p<r){
        int q=(p+r)/2;
        mergeSort(a,size,p,q,comp);
        mergeSort(a,size,q+1,r,comp);
        merge(a,size,p,q,r,comp);
    }
}
```

程序 3-6 实现算法 3-3 的 MERGE-SORT 过程的 C 函数 mergeSort

对程序 3-6 的说明如下。

(1) mergeSort 函数有 5 个参数：其中，a、p、r 对应于 MERGE-SORT 过程的参数 A、p、r。注意，a 的类型是 **void** *，这意味着该函数能对任何类型的数组做归并排序操作。参数 size 确定 a 中元素的存储长度，参数 comp 确定 a 中元素大小比较规则。

(2) C 语言中整型数据的除法运算对整数是封闭的，所以对正整数 a 和 b，a/b=$\lfloor a/b \rfloor$。于是，程序中的第 4 行 q=(p+r)/2 对应于算法 3-3 中第 2 行的赋值操作 $q \leftarrow \lfloor (p+r)/2 \rfloor$。

(3) 程序代码与算法伪代码十分近似，读者可对照研读。

为便于重用，将程序 3-6 中的代码存储在 utility 文件夹中的源文件 mergesort.c 中，并将此函数的原型声明存储在头文件 mergesort.h 中。

2) **链表版本**

利用第 2 章中引入的链表 LinkedList 以及程序 3-2 中的合并过程 MERGE 的链表实现版本 listMerge，可将过程实现为如下链表版本。

```
1 void listMergeSort(LinkedList *A,ListNode *p,ListNode *r){
3     if(p!=r&&p!=r->next&&p->prev!=r){
4         int n=distance(p,r);
5         ListNode *q=p;
6         q=advance(q,n/2);
7         listMergeSort(A,p,q);
8         listMergeSort(A,q->next,r);
9         listMerge(A,p,q,r);
10    }
11 }
```

程序 3-7　实现算法 3-3 的 C 函数的链表版本 listMergeSort

对程序 3-7 的说明如下。

(1) 和算法过程一样，函数 listMergeSort 有 3 个参数。参数 A 表示序列 A[p..r]，参数 p、r 分别是指向序列的首元素和尾元素的指针。

(2) 与程序 3-6 中函数 mergeSort 相比，对 p<r 的判断，由于 p、r 不再是表示数组元素的下标，而是指向结点的指针，所以此条件改变为 p!=r && p!=r->next && p->prev!=r。变量 q 定义成了指向链表结点的指针类型而非表示数组下标的整型。为了使 q 指向 p、r 之间的中点，先调用函数 distance，计算 p、r 之间的结点数 n(第 4 行)。该函数的定义很简短：

```
int distance(ListNode *p,ListNode *r){
    int dist=1;
    while(p!=r){
        dist++;
        p=p->next;
    }
    return dist;
}
```

为便于重用，将此函数及其原型声明分别添加到 list.c 和 list.h 中。

然后,令 q 从 p 开始(第 5 行),调用函数 advance 让 q 往后行进 n/2 个结点。该函数的定义也很简短:

```
ListNode * advance(ListNode * i,int d){
    int distance=d>=0?d:-d,k=1;
    while(k<distance){
        i=(d>=0)?i->next: i->prev;
        k++;
    }
    return i;
}
```

它将传递给它的结点指针参数 i 在链表中向前(d<0)移动或向后(d>0)移动|d|个结点。为便于重用,将此函数及其原型声明分别添加到 list.c 和 list.h 中。

(3) 将程序 3-6 中所有对数组元素的访问替换成对链表结点的访问,下标的自增换成取下一个结点的操作,就形成了程序 3-7 中相应的代码。

为便于重用,将函数 listPartition 的定义代码存储为 utility 文件夹内的源文件 listmergesort.c,并将该函数的原型声明存储为头文件 listmergesort.h 中。

3.2.2 快速排序算法

1. 算法的伪代码描述

快速排序也是一个典型的分治算法:和归并排序一样,将 $A[p..r]$划分成 $A[p..q]$和 $A[q+1..r]$两部分,但不是对分($q=\lfloor(p+r)/2\rfloor$),而是利用 3.1.2 节中的 PARTITION 过程使得 $A[p..q]$中的元素值小于 $A[q+1..r]$中的元素值。递归地解决 $A[p..q]$和 $A[q+1..r]$后就可省略合并过程了。

```
QUICKSORT(A,p,r)
if p<r
   then q←PARTITION(A,p,r)
        QUICKSORT(A,p,q-1)
        QUICKSORT(A,q+1,r)
```

算法 3-4 快速排序算法

2. 算法的运行时间

仔细考察两次递归所处理的子问题,如前所述 $A[p..q]$和 $A[q+1..p]$的规模不必相同。

最好情形发生在每次调用 PARTITION 过程总是将 $A[p..r]$分成两个长度相同的子序列 $A[p..q]$和 $A[q+1..r]$,其中 $q-p+1=r-q$。此时,两次递归的运行时间为 $2T(n/2)$,每次划分的运行时间为 $\Theta(n)$(见 3.1.2 节),这样将得到一个与归并排序运行时间相同的递归方程:

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+\Theta(n) & n>1\end{cases}$$

运用定理 3-1 知，QUICKSORT 的最好情形运行时间为 $\Theta(n\lg n)$。

最坏情形发生于每次调用 PARTITION 过程总是将 $A[p..r]$分成 $A[p]$和 $A[p+1..r]$（或总是分成 $A[p..r-1]$和 $A[r]$），这样将得到一个递归方程：

$$T(n)=T(n-1)+\Theta(n)$$

其中 $\Theta(n)$表示划分时间，$T(n-1)$表示递归所需时间，为方便计算，用 n 表示 $\Theta(n)$，运用递推法：

$$\begin{aligned}T(n)&=T(n-1)+n\\&=T(n-2)+(n-1)+n\\&\ \ \vdots\\&=1+2+\cdots+(n-1)+n\\&=n(n+1)/2=\Theta(n^2)\end{aligned}$$

得到该方程的解为 $\Theta(n^2)$。

如此看来，快速排序不如归并排序好。但是，这存在着一个微妙之处：快速排序的平均情形运行时间更接近于最好情形时间。例如，假定划分算法总是产生 9∶1 比例的划分，这使得开始时看起来划分很不平衡。得到运行时间的递归式：

$$T(n)\leqslant T(9n/10)+T(n/10)+cn$$

其中已经显式地表示了隐藏在 $\Theta(n)$背后的常数因子。图 3-5 展示了此递归式的递归树。注意树的每一层具有代价 cn，直至遇到边界条件深度 $\log_{10}n=\Theta(\lg n)$，此后各层的代价至多为 cn。递归于深度 $\log_{10/9}n=\Theta(\lg n)$处终止。于是，快速排序的总代价为 $O(n\lg n)$。因此，一个在每一层递归都按 9∶1 的比例进行分裂，初始时很不平衡的快速排序运行于 $O(n\lg n)$时间与在中心分裂的渐近相同。事实上，甚至是 99∶1 的比例也得出 $O(n\lg n)$的运行时间。原因是按任何常数比例分裂都导致一棵深度为 $\Theta(\lg n)$的树，而每一层的代价为 $O(n)$。所以无论以什么比例分裂运行时间都是 $O(n\lg n)$。更精细的分析可以得到快速排序的平均情形运行时间为 $\Theta(n\lg n)$。

根据对快速排序的这一观察，人们设法控制算法的行为——每次划分随机地在 $A[p..r]$中选取一个元素作为划分基准，以此来获得平均情形：

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

算法 3-5　划分算法的随机版本

然后在排序过程中调用 RANDOMIZED-PARTITION 对 $A[p..r]$进行划分。

图 3-5 所示为 PARTITION 总是产生 9∶1 的分裂的 QUICKSORT 的运行时间递归树，得出运行时间为 $\Theta(n\lg n)$。右边显示的是每一层的代价。每一层代价包含了隐藏于 $\Theta(n)$内的常数 c。

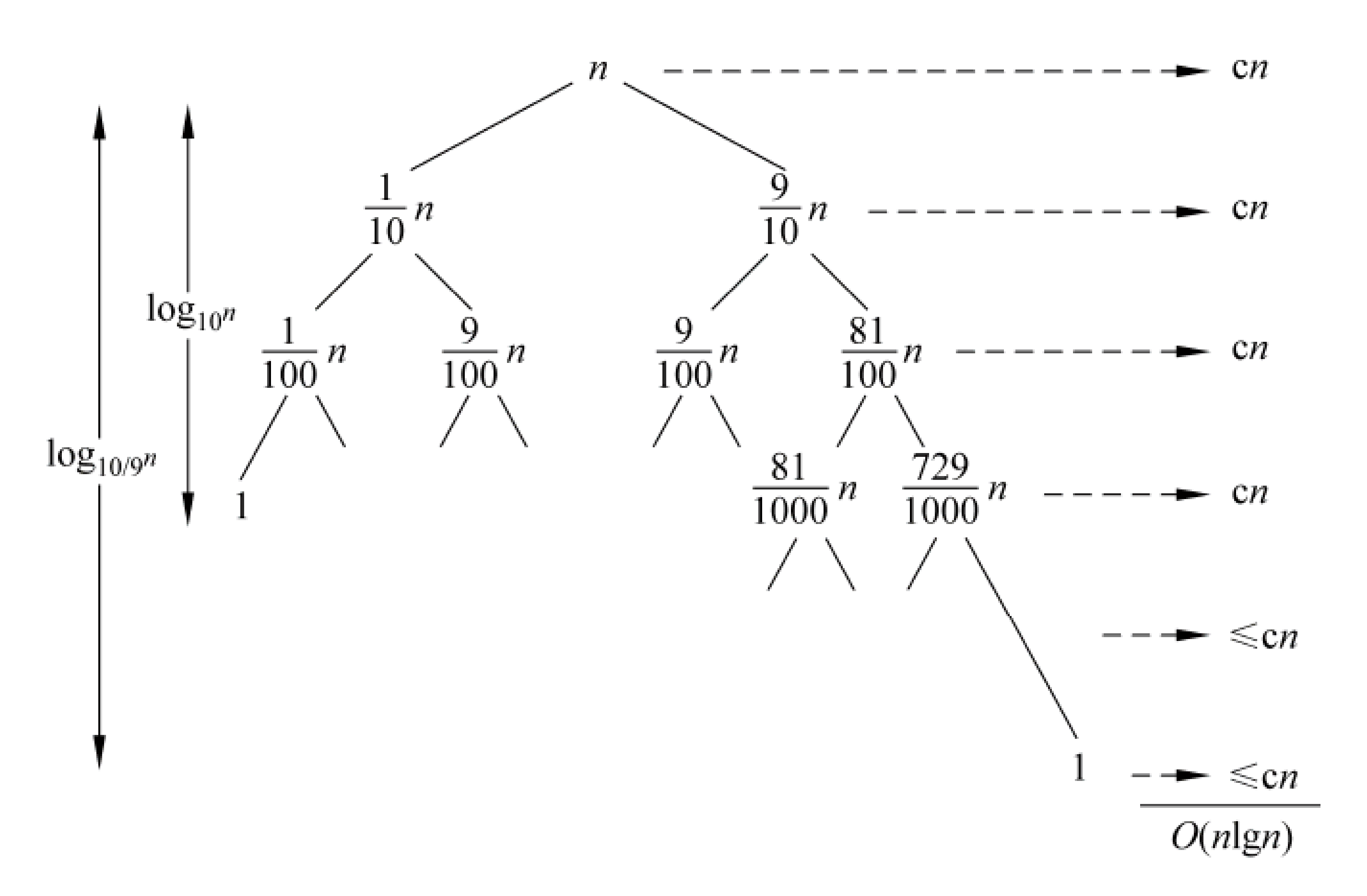


图 3-5　快速排序运行时间递归树

```
RANDOMIZED-QUICKSORT(A, p, r)
if p<r
  then q←RANDOMIZED-PARTITION(A, p, r)
       RANDOMIZED-QUICKSORT(A, p, q-1)
       RANDOMIZED-QUICKSORT(A, q+1, r)
```

算法 3-6　快速排序算法的随机版本

以此来获得平均运行时间 $\Theta(n\lg n)$。在算法中人为地使用随机数发生器来影响算法的行为是一种提高算法运行效率的方法,使用这种方法设计的算法称为**随机算法**。随机数发生器在大多数高级程序设计语言中都作为库函数提供给程序员使用,在 C 语言中有定义在 stdlib. h 中的库函数 rand(),它返回一个 0～RAND_MAX 之间的随机数值。

3. 程序实现

首先利用 3.1.2 节中的 partition 函数实现算法 3-7。

```
1 int randomNumber(int p,int q){
2     return p+(int)((double)(q-p) * rand()/(RAND_MAX));
3 }
4 int randmizedPartition(void *a,int size,int p,int r,int(*comp)(void *,void *)){
5     int i=randomNumber(p,r);
6     swap(a+r*size,a+i*size,size);          /* a[r]↔a[i] */
7     return partition(a,size,p,r,comp);
8 }
```

程序 3-8　实现算法 3-7 的 C 函数的数组版本

对程序 3-8 的说明如下。

(1) 第 1～3 行定义的函数 randomNumber 调用库函数 rand()计算介于两个参数 p、q 之间的一个随机整数,并将其返回。该函数代码存储为 utility 文件夹中的源文件 randomn. c。

(2) 第 4～8 行定义的函数 randmizedPartition 实现算法 3-5 中的 RANDOMIZED-PARTITION 过程的数组版本。程序代码与伪代码十分接近。将这个函数的定义添加到源文件 partition.c 中，并将它的原型声明添加到头文件 parttion.h 中。

(3) 实现算法 3-5 中的 RANDOMIZED-PARTITION 过程的链表版本如下：

```
ListNode* rndListPartition(LinkedList *A,ListNode *p,ListNode *r){
    int d=distance(p,r),t=randomNumber(0,d);
    ListNode *i=advance(p,t);
    swap(i->key,r->key,size);
    return listPartition(A,p,r);
}
```

程序 3-9　实现算法 3-5 的 C 函数链表版本

该函数先计算链表中结点 p 到结点 r 之间所含结点的个数(调用 distance 函数，见 3.2.1 节)d，产生 0～d 之间的随机数 t，找到链表中从 p 起的第 t 个结点(调用 advance 函数，见 3.2.1 节)i，交换结点 i、r 后调用 listPartition 对从 p 到 r 进行划分操作。

将这个函数的定义添加到源文件 listpartition.c 中，并将它们的原型声明添加在头文件 listpartition.h 中。做好这些准备工作后，实现算法 3-6 中过程 RANDOMIZED-QUICKSORT。

```
1 void quickSort(void* a,int size,long p,long r,int(*comp)(void *,void *)){
2     if(p<r){
3         long q=randmizedPartition(a,size,p,r,comp);
4         quickSort(a,size,p,q-1,comp);
5         quickSort(a,size,q+1,r,comp);
6     }
7 }
8 void listQuickSort(LinkedList *A,ListNode *p,ListNode *r){
9     if(p!=r){
10        ListNode *q=rndListPartition(A,p,r);
11        listQuickSort(A,p,q!=p?q->prev:p);
12        listQuickSort(A,q!=r?q->next:r,r);
13    }
14 }
```

程序 3-10　实现算法 3-6 的 C 源代码

对程序 3-10 的说明如下。

(1) 第 1～7 行定义的函数 quickSort 是实现算法 3-6 的数组版本，它含有 5 个参数。其中，a 表示数组，size 表示数组中元素的存储宽度，p、r 为数组 a 的首、尾元素下标，comp 表示数组 a 中元素之间大小比较规则。程序代码与伪代码十分接近。将这个函数的定义存储为 utility 文件夹中的源文件 quicksort.c，并且将它们的原型声明存储为头文件 quicksort.h，以备重用。

(2) 第 8～14 行定义的函数 listQuickSort 是实现算法 3-6 的链表版本，它含有 3 个参

数。其中,A 表示序列 A[p..r],p、r 为指向链表中带排序部分的首、尾结点指针。

(3) 算法中以 p<r 来判断 a[p]、a[r]之间是否存在多于 1 个的元素。由于此处 p、r 不再是表示数组元素的下标,而是指向结点的指针,所以此条件需改变为 p! =r。变量 q 也定义成了指向链表结点的指针类型而非表示数组下标的整型,对第 10 行计算出来的 q,若 q==p,则对左半部分的递归传递的第 3 个参数就是 p,否则传递 q->next。类似地,若 q==r,则对右半部分的递归第 3 个参数传递 r,否则传递 q->prev。将这个函数的定义存储为 utility 文件夹中的源文件 listquicksort.c,并且将它们的原型声明存储为头文件 listquicksort.h。

3.2.3 序统计与选择问题

1. 问题的描述

n 个元素集合的第 i 个**序统计**是第 i 小元素。例如,集合的**最小**元素就是第一序统计($i=1$),而**最大**元素就是第 n 序统计($i=n$)。**中值**,也就是集合的“中途点”。当 n 为奇数时,中值是唯一的,发生在 $i=(n+1)/2$ 处。而当 n 是偶数时,有两个中值,发生在 $i=n/2$ 和 $i=n/2+1$ 处。于是,不管 n 的奇偶性,中值总是发生在 $i=\lfloor (n+1)/2 \rfloor$(**低中值**)处和 $i=\lfloor (n+1)/2 \rfloor$(**高中值**)处。为方便计,词语“中值”指的是低中值。

此处研究从 n 个互不相同数的集合中选取第 i 序统计的问题,即**选择问题**。尽管所做的工作都可以扩展到集合中有重复元素的情形,为方便计仍然假定集合中的数各不相同,且约定用线性表表示集合 A。选择问题可以形式化地说明如下。

输入:线性表 $A[p..r]$中 $n(n=p-r)$个(各不相同)的元素 $A[p..r]$以及满足 $1\leqslant i\leqslant n$ 的数 i。

输出:元素 $x\in A[p..r]$,它恰比 $A[p..r]$中的其他 $i-1$ 个元素大。

选择问题可以在 $O(n\lg n)$时间内解决,因为可以用归并排序将各数排序然后直接指出输出数组中的第 i 个元素,然而却有更快的算法。

2. 算法的伪代码描述

解决选择问题的一个算法类似快速排序算法的思想,为了在序列 $A[p..r]$中找出第 i 小的元素,先调用 RANDOMIZED-PARTITION 过程将 A 划分成 $A[p..q]$和 $A[q+1..r]$,使得 $A[p..q]$中的元素值不超过 $A[q]$,$A[q+1..r]$中的元素值大于 $A[q]$。若 $i=q-p+1$,则 $A[q]$即为所求;若 $i<q-p+1$,则问题转化为在 $A[p..q-1]$中找第 i 小元素;若 $i>q-p+1$,则问题转化为在 $A[q+1..r]$中找第 $i-q+p-1$ 小元素。

```
SELECT(A,p,r,i)
if p=r
   then return A[p]
q←RANDOMIZED-PARTITION(A,p,r)
k←q-p+1
if i=k                          ▷基准元素即为答案
   then return A[q]
```

```
else if i<k
    then return SELECT(A,p,q-1,i)
    else return SELECT(A,q+1,r,i-k)
```

算法 3-7　解决选择问题的 SELECT 算法过程

3. 算法的运行时间

与快速排序算法的分析相似,此过程最坏情形发生在每次划分都得到序列的前端或末端,运行时间为 $T(n)=T(n-1)+n$。利用迭代法,很容易计算出 $T(n)=\Theta(n^2)$。最好情形发生在每次划分都得到中点,与快速排序不同的是,此处只需要处理两个子序列之一,所以运行时间为 $T(n)=T(n/2)+n$,根据本章定理 3-1 可知,$T(n)=\Theta(n)$。平均情形时间也与快速排序算法相似,与最好情形运行时间相同,为 $\Theta(n)$。

4. 程序实现

利用程序 3-8 和程序 3-9 定义的函数 randmizedPartition 和 rndListPartition,可实现如下的 SELECT 算法的数组版本和链表版本。

```
void* select(void* a,int size,int p,int r,int i,int(*comp)(void *,void *)){
    int q,k;
    if(p==r)
        return (char*)a+p*size;          /*返回 a[p]*/
    q=randmizedPartition(a,size,p,r,comp);
    k=q-p;
    if(k==i)
        return (char*)a+q*size;          /*返回 a[q]*/
    if(i<k)
        return select(a,size,p,q-1,i,comp);
    return select(a,size,q+1,r,i-k-1,comp);
}
ListNode* listSelect(LinkedList *a,ListNode *p,ListNode *r,int i){
    ListNode *q;
    int k;
    if(p==r)
        return p;
    q=rndListPartition(a,p,r);
    k=distance(p,q)-1;
    if(k==i)
        return q;
    if(i<k)
        return listSelect(a,p,q!=p?q->prev:p,i);
    return listSelect(a,q!=r?q->next:r,r,i-k-1);
}
```

程序 3-11　实现算法 3-7 的 C 函数

对程序 3-11 的说明如下。

(1) 第 1～12 行定义的是实现算法 3-7 的数组版本函数 select。与算法过程相比,参数除了表示数组的 a、起点 p、终点 r 和序号 i 以外,还多了表示数组元素存储宽度的 size 和比较元素大小的函数指针参数 comp。这是因为实现的是利用 **void** * 的通用代码。

(2) 伪代码中序统计操作中,序号从 1 开始,符合其数组下标的编排规则。在 C 语言中,数组的下标是从 0 开始编排的。为不减弱代码的可读性,序号也从 0 开始编排。这样就需要对程序中表示 p、q 间包含的元素个数的 k 做相应的调整。把此函数的定义及其原型声明分别存储为 utility 文件夹内的源文件 select.c 和头文件 select.h,以备重用。

(3) 第 13～25 行定义的是实现算法 3-7 的链表版本函数 listSelect。该函数的参数与算法过程参数是一致的。这是因为已经把链表中元素的存储宽度和元素间的比较规则都已经封装在了链表内(见 2.1.3 节)。把此函数的定义及其原型声明分别存储为 utility 文件夹内的源文件 listselect.c 和头文件 listselect.h。

3.3 排序问题的讨论

3.3.1 排序的性质

1. 就地排序

在排序过程中,任何时刻至多有常数个元素存储于序列之外,则排序过程称为就地排序过程。按此概念,快速排序是就地排序过程,这是因为,在 QUICK-SORT 的运行过程中,任何时刻,序列中仅有最后一个元素 $A[r]$存储于序列之外的变量 x 中。而归并排序算法不是就地排序过程,因为,在 MERGE-SORT 的运行过程中,每一层次的递归都需要调用 MERGE 过程。MERGE 过程中,需要设置辅助序列 L 和 R,它们需要占据 $\Theta(n)$的空间,其中 n 表示序列所含元素个数。

2. 稳定性

在排序过程中,如果具有相同值的元素在输出数组中的顺序与其在输入数组中的相同,则该算法称为是**稳定**的。按此概念,研究所讨论过的插入排序、归并排序、快速排序和堆排序算法不难作出如下断言。

定理 3-2 归并排序是稳定排序,而快速排序不是稳定排序。

在归并排序算法 3-3 中,对两个子序列递归后需要调用算法 3-1 的 MERGE 过程加以合并。在 MERGE 过程中,仅当 $L[i]>R[j]$时,在原序列中后面的元素 $R[j]$才会发生位置前移到原序列中前面的元素 $L[i]$之前的变化。所以,数值相同的元素之间不会发生位置的相对变化。根据稳定性概念可知归并排序算法是稳定的。对于快速排序,只要考察划分算法就可知道它不是稳定的。例如,对图 3-6(a)的序列进行 PARTITION 操作得到图 3-6(b)的序列。注意原来位于 7 后面的元素 $7'$换位到了它的前面。

在一些应用中,排序的稳定性是很重要的。例如,当排序条件包含两个关键字时,往往

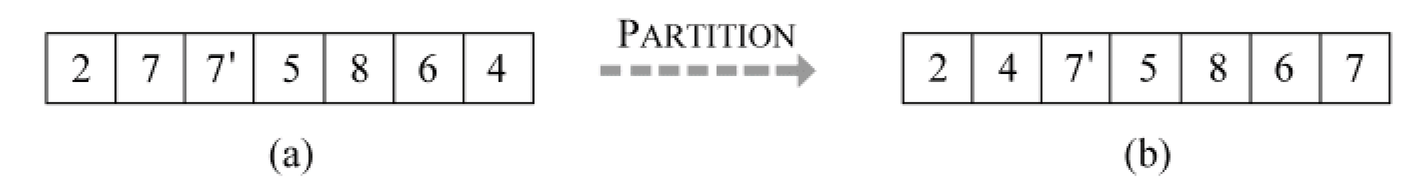


图 3-6　PARTITION 过程的不稳定性

要求在第一个关键字相等的情况下，要计算第二个关键字的某种比较关系，这就需要对第一个关键字排序是稳定的。

3.3.2　比较型排序算法的时间复杂度

前面讨论了解决排序问题的两种排序算法：归并排序和快速排序。这两个算法有一个共同的特性：排序顺序的确定仅基于序列中元素间的比较，将这些排序算法称为**比较排序**。

1. 决策树

比较排序可以抽象地视为一棵**决策树**。一棵决策树是一棵满二叉树[①]，它表示了一个具体排序算法对给定规模输入的操作时各元素之间的比较。算法中的控制、数据移动和其他所有的方面都被忽略。为使行为简洁，就下列简单的插入排序算法展开讨论。

```
INSERTION-SORT (A)
for j←2 to length[A]
  do key←A[j]
     ▷将 A[j]插入到排好序的序列 A[1..j−1]中
     i←j−1
     while i>0 and A[i]>key
       do A[i+1]←A[i]
          i←i−1
     A[i+1]←key
```

算法 3-8　解决排序问题的 INSERTION-SORT 算法

图 3-7 展示了插入算法对一个三元素的输入序列的操作。

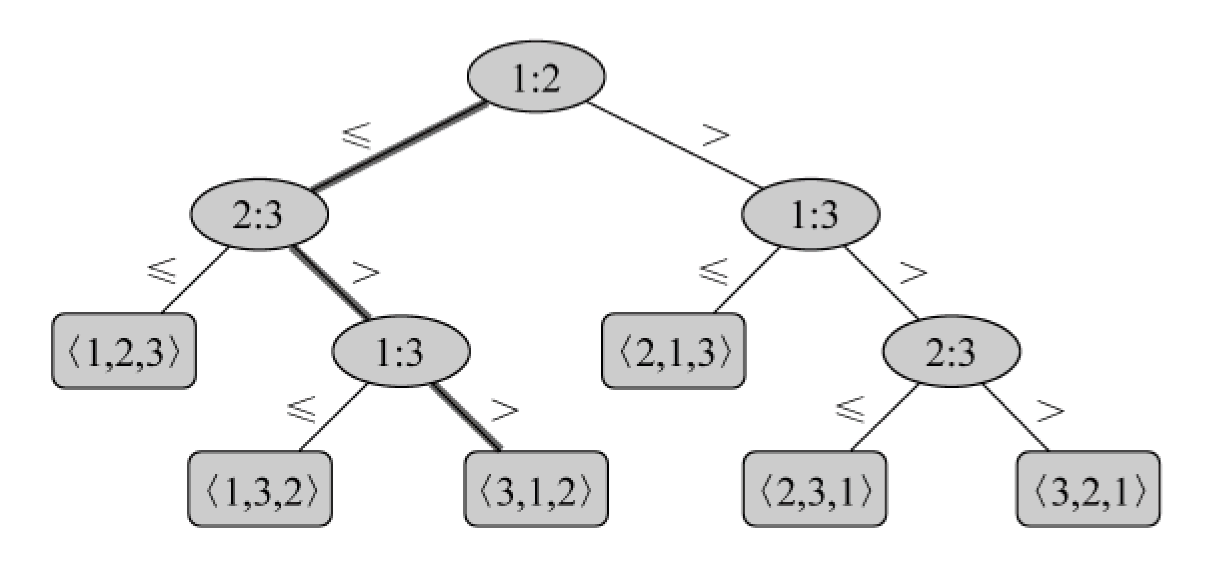


图 3-7　对 3 个元素进行插入排序的决策树

图 3-7 所示为对 3 个元素进行插入排序的决策树。标注为 $i:j$ 的内结点指出 a_i 和 a_j

① 满二叉树指的是具有下述性质的二叉树：树中的每个内点都具有两个孩子。

之间的一次比较。由排列 $<\pi(1),\pi(2),\cdots,\pi(n)>$ 表示的一片叶子指示出一个排序 $a_{\pi(1)}\leqslant a_{\pi(2)}\leqslant\cdots\leqslant a_{\pi(n)}$。阴影路径指示出对输入序列 $<a_1=6,a_2=8,a_3=5>$ 排序所做的决策;叶子处的排列 $<3,1,2>$ 指示出排好序的顺序为 $a_3=5,a_1=6,a_2=8$。输入元素共有 $3!=6$ 种可能的排列,所以决策树中共有 6 片叶子。

在一棵决策树中,每一个内点都标注为 $i:j$,其中 i 和 j 取自范围 $1\leqslant i,j\leqslant n$,而 n 是输入序列中数的个数。每一片叶子用一个排列 $<\pi(1),\pi(2),\cdots,\pi(n)>$ 加以标注。排序算法的执行对应于决策树中一条从根到一片叶子的路径。在每一个内点处做一次比较。左子树指示出 $a_i\leqslant a_j$ 后来的比较,右子树指示出 $a_i>a_j$ 后来的比较。当来到一片叶子时,排序算法已经完成了排序 $a_{\pi(1)}\ a_{\pi(2)}\cdots\ a_{\pi(n)}$。由于任一正确的排序算法必能产生其输入序列的每一个排列,比较排序正确的必要条件是 n 个元素的 $n!$ 个排列中的每一个必须作为一片叶子出现在决策树中,并且从根起到这些叶子中的每一片都有一条对应于该比较排序的一次确切的执行的路径(将这样的叶子称为从根"可达"的)。于是,将仅考虑每个排列都显示为可达叶子的决策树。

从决策树的根起,到其任意一片可达的叶子的最长路径的长度表示了对应的排序算法的最坏情形所执行的比较次数。因此,对于给定的排序算法最坏情形的比较次数等于其决策树的高度,输入的每一个排列作为决策树中的一片可达叶子。所有决策树的高度的下界就是任一比较排序算法运行时间的下界。由于高度为 h 的二叉树至多有 2^h 片叶子,所以对有 $n!$ 片可达叶子的满二叉树而言,有 $n!\leqslant 2^h$,对其取对数,蕴涵着 $h\geqslant\lg(n!)\geqslant n\lg n$。① 于是可得到如下定理。

定理 3-3 任一比较排序算法在最坏情形下至少需要做 $n\lg n$ 次比较。

这意味着任何一种比较型的排序算法,其时间复杂度至少是 $n\lg n$。由此可见,理论上归并排序是最优的排序算法,因为它们在最坏情形下的运行时间是 $\Theta(n\lg n)$。

2. 计数排序

若排序过程中,决定元素前后排列位置并不基于元素间的大小比较,则算法的最坏情形运行时间的下限不受定理 3-2 的约束。例如,下列讨论的计数排序算法就是一个典型的不是基于比较的排序算法。

1) 算法描述与分析

计数排序假设 n 个输入元素的每一个是 0 到 k 中的一个整数,其中 k 为一个整数。当 $k=O(n)$ 时,排序时间为 $\Theta(n)$。

计数排序的基本思想是对每一个输入元素 x 确定比 x 小的元素个数。这一信息可以用来将 x 直接置于其在输出数组中的位置。例如,若有 17 个元素小于 x,则 x 应在输出数组的第 18 个位置上。这一方案在有若干个元素具有相同值时,要做一点修改,因为不想把它们放在一个位置上。

在计数排序的代码中,假设输入是一个数组 $A[1..n]$,且其长度 $length[A]=n$。还需要两个数组:数组 $B[1..n]$ 存储排好序的输出,数组 $C[0..k]$ 提供一个临时工作空间。

① 根据斯特林公式 $n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$,取对数可得。

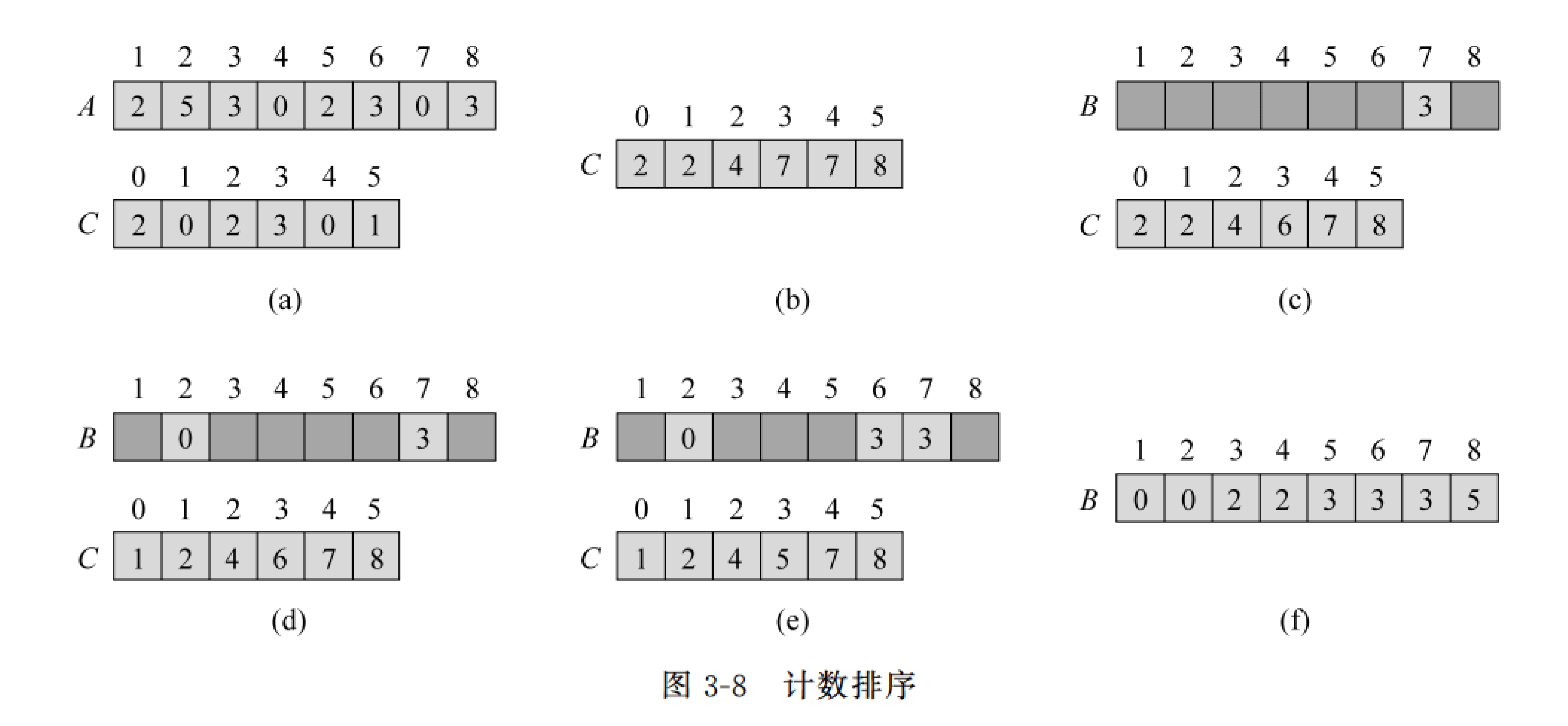


图 3-8　计数排序

图 3-8 所示为 COUNTING-SORT 对数组 $A[1..8]$的操作，其中 A 的每个元素都是不超过 $k=5$ 的非负整数。图 3-8(a)为数组 A 和第 4 行后的数组 C。图 3-8(b)为第 7 行后的数组 C。图 3-8(c)～图 3-8(e)为输出数组 B 和分别为第 9～11 行的循环的第一、二、三次重复后的辅助数组 C。数组 B 中仅浅阴影元素才是被填入的。图 3-8(f)为最后排好序的数组 B。

```
COUNTING-SORT(A,B,k)
1 for i←0 to k
2    do C[i]←0
3 for j←1 to length[A]
4    do C[A[j]]←C[A[j]]+1
5 ▷C[i]包含了等于 i 的元素个数
6 for i←1 to k
7    do C[i]←C[i]+C[i-1]
8 ▷C[i] 包含了小于或等于 i 的元素个数
9 for j←length[A] downto 1
10    do B[C[A[j]]]←A[j]
11       C[A[j]]←C[A[j]]-1
```

算法 3-9　计数排序过程 COUNTING-SORT

计数排序过程如下。在第 1 行和第 2 行的 **for** 循环做了初始化后，在第 3 行和第 4 行的 **for** 循环中检测每一个输入元素。若一个输入元素的值是 i，增加 $C[i]$。于是，第 4 行后，对 $i=0,1,\cdots,k$，$C[i]$存储了等于 i 的输入元素个数。在第 6 行和第 7 行，对每一个 $i=0,1,\cdots,k$，通过求数组 C 的相邻项的和来确定有多少个元素小于或等于 i。

最后，在第 9 行和第 11 行的 **for** 循环中，将元素 $A[j]$置入其在输出数组 B 中的正确的排序位置。若所有的 n 个元素是各不相同的，则当首次进入第 9 行时，对每一个 $A[j]$，值 $C[A[j]]$就是 $A[j]$在输出数组 B 中的位置。由于各元素不必不同，每次把 $A[j]$置入数组 B 后从 $C[A[j]]$中减值。对 $C[A[j]]$的减值使得下一个其值等于 $A[j]$的元素可直接到达输出数组中 $A[j]$前的位置。

计数排序需要多少时间呢？第 1 行和第 2 行的 **for** 循环耗时 $\Theta(k)$，第 3 行和第 4 行的

for 循环耗时 $\Theta(n)$，第 6 行和第 7 行的 **for** 循环耗时 $\Theta(k)$，而第 9～11 行的 **for** 循环耗时 $\Theta(n)$。于是，总耗时 $\Theta(k+n)$。实践中，当 $k=O(n)$ 时，常使用计数排序，此时的运行时间是 $\Theta(n)$。

计数排序突破了定理 3-2 的 $n\lg n$ 下界，这是因为它不是比较排序。事实上，在代码中没有发生任何元素间的比较。计数排序是用元素的确切值来将元素编入一个数组的。不是比较排序模型就不能应用 $\Omega(n\lg n)$ 的排序下界。计数排序是稳定的，但不是就地排序。

2）**程序实现**

```
1 #include<stdlib.h>
2 void countSort(unsigned *a,int n){
3     unsigned *b=(unsigned *)malloc((n)*sizeof(unsigned)),*c,k=a[0];
4     int i;
5     for(i=1;i<n;i++)                  /*找出 a 中最大值 k*/
6         if(a[i]>k)
7             k=a[i];
8     c=(unsigned *)malloc((k+1)*sizeof(unsigned));
9     for(i=0;i<=k;i++)                 /*数组 c 清 0*/
10        c[i]=0;
11    for(i=0;i<n;i++)                  /* c[i]包含了等于 i 的元素个数*/
12        c[a[i]]++;
13    for(i=1;i<=k;i++)                 /* c[i] 包含了小于或等于 i 的元素个数*/
14        c[i]=c[i-1]+c[i];
15    for(i=n-1;i>-1;i--){
16        b[c[a[i]]-1]=a[i];
17        c[a[i]]--;
18    }
19    memcpy(a,b,n*sizeof(unsigned));
20    free(b);free(c);
21 }
```

程序 3-12　实现算法 3-9 的 C 源代码

对程序 3-12 的说明如下。

（1）函数 countSort 有 2 个参数，a 为待排序的数组，n 指出 a 中所具有的元素个数。

（2）第 3 行为数组 b 分配 n 个存储单元，用来存放对 a 排序的结果。算法 3-9 中的 C 是一个长度足够大的数组，要能包含 A 中最大值作为下标的元素。为了合理地分配空间，第 5～7 行计算数组 a 中的最大值，记录在变量 k 中。第 8 行为数组 c 分配 k 个存储单元。

（3）第 8～18 行并排的 4 个 **for** 循环实现算法 3-9 中对应的 4 个 **for** 循环。程序代码与伪代码十分接近。第 19 行将排序结果 b 复制到 a 中，以符合实用中对数组 a 的排序结果仍然在 a 中的习惯。

为便于重用，将程序 3-9 存储为 utility 文件夹中的源文件 countsort.c，并将该函数的原型声明存储为头文件 countsort.h。

3.3.3　应用

排序算法有广泛的应用。在第 1 章就看到过，数据经过排序后，利用二分查找法查找指

定数据项要比普通线性查找效率高得多。以后会看到，很多解决经典问题的算法都要对数据进行包含排序在内的预处理。解决下列的环法自行车游问题就需要用到排序。

1. 环法自行车游

Tour De France

Description

A racing bicycle is driven by a chain connecting two sprockets. Sprockets are grouped into two clusters: the front cluster (typically consisting of 2 or 3 sprockets) and the rear cluster (typically consisting of between 5 and 10 sprockets). At any time the chain connects one of the front sprockets to one of the rear sprockets. The drive ratio—the ratio of the angular velocity of the pedals to that of the wheels—is $n : m$ where n is the number of teeth on the rear sprocket and m is the number of teeth on the front sprocket. Two drive ratios $d_1 < d_2$ are adjacent if there is no other drive ratio $d_1 < d_3 < d_2$. The *spread* between a pair of drive ratios $d_1 < d_2$ is their quotient: d_2/d_1. You are to compute the maximum spread between two adjacent drive ratios achieved by a particular pair of front and rear clusters. You may assume that no cluster has more than 10 sprockets and that no gear has fewer than 10 or more than 100 teeth.

Input

Input consists of several test cases, followed by a line containing 0. Each test case is specified by the following input.

- f: the number of sprockets in the front cluster.
- r: the number of sprockets in the rear cluster.
- f integers, each giving the number of teeth on one of the gears in the front cluster.
- r integers, each giving the number of teeth on one of the gears in the rear cluster.

Output

For each test case, output the maximum spread rounded to two decimal places.

Sample Input

```
2 4
40 50
12 14 16 19
0
```

Sample Output

```
1.19
```

1）问题描述与分析

自行车的驱动系统包含很多齿轮。齿轮通常分成两组：前端齿轮和后端齿轮。自行车就是通过连接一个前端齿轮和一个后端齿轮来驱动的。设有 f 个前端齿轮，r 个后端齿轮，每个前端齿轮 i 有 $m[i]$ 个齿，$i=1,2,\cdots,f$。每个后端齿轮 j 有 $n[j]$ 个齿，$j=1,2,\cdots,r$。所

以共有 $f \cdot r$ 个可能的驱动率 $d[k]=n[j]:m[i]$,$k=1,2,\cdots,r\cdot f$,$j=1,2,\cdots,r$,$i=1,2,\cdots,f$。假定 d 排好序,这样数组 d 中两个相邻元素满足题中若 $d_1<d_2$ 不存在 d_3 使得 $d_1<d_3<d_2$ 的条件。由此,可计算出 $spread[1..f\cdot r-1]$数组:$spread[k]=d[k+1]/d[k]$,$k=1,2,\cdots,r\cdot f-1$。所以,本问题可以形式化如下。

输入:数组 $m[1..f]$、$n[1..r]$。

输出:根据数组 m 和 n 计算出来的 $spread[1..f\cdot r-1]$数组中的最大值。

2) **算法描述**

```
TOUR-DE-FRANCE(m,n)
f←length[m]
r←length[n]
k←1
for j←1 to r
  do for i←1 to f
        do d[k]←n[j]/m[i]
           k←k+1
SORT(d)
for k←1 to r · f-1
   do spread[k]←d[k+1]/d[k]
return SELECT(spread,r · f-1)
```

算法 3-10 解决 Tour De France 问题的算法过程

其中,过程 SELECT(A,p,r,i)是 3.2.3 节中描述的选择序列 $A[p..r]$中第 i 小元素的算法 3-7。

3) **程序实现**

为节省篇幅,此处仅列出实现算法 3-10 的 tourDeFrance 函数,调用此函数解决 Tour De France 问题的主调函数与 tourDeFrance 函数一起存储在文件夹 chap03/Tour De France 的源文件 tourdefrance.c 中,读者可打开该文件研读。

```
double tourDeFrance(int *m,int *n,int f,int r){
 double *d=(double*)malloc(f*r*sizeof(double)),          /*为数组 d 分配空间*/
  *spread=(double*)malloc((f*r-1)*sizeof(double)),       /*为数组 spread 分配空间*/
 result;                                                 /*用来存放计算结果*/
 int i,j,k=0;
 assert(d&&spread);
 for(j=0;j<r;j++)                                        /*计算数组 d*/
     for(i=0;i<f;i++)
       d[k++]=(double)n[j]/m[i];
  quickSort(d,sizeof(double),0,f*r-1,doubleGreater);     /*对数组 d 按升序排序*/
  for(k=0;k<f*r-1;k++)                                   /*计算数组 spread*/
      spread[k]=d[k+1]/d[k];
  free(d);
  result=*((double*)select(spread,sizeof(double),0,f*r-2,
```

```
         f * r-2,doubleGreater));  /* 在数组 spread 中选择最大者作为计算结果 */
15    free(spread);
16    return result;
17 }
```

程序 3-13　实现算法 3-10 的 C 源代码

对程序 3-13 的说明如下。

(1) 函数 tourDeFrance 有 4 个参数，m、n 表示数组。由于 C 语言的数组没有长度属性，所以参数 f、r 分别表示这两个数组的元素个数。该函数返回计算结果：spread 数组中的最大者，这是一个带小数的实数。

(2) 第 2～5 行声明程序中所需的各个变量。

(3) 第 7～9 行的两重 **for** 循环嵌套对应算法 3-10 中第 4～7 行的 **for** 循环嵌套，完成对数组 d 的计算。第 10 行调用 3.2.2 节开发的对数组的快速排序函数 quickSort 对数组 d 进行升序排序。第 11 行和第 12 行的 **for** 循环对应算法 3-10 中第 9 行和第 10 行的循环，计算数组 spread。第 14 行调用 3.2.3 节中开发的实现算法 3-7 的 SELECT 过程的函数 select，找到数组 spread 中的最大元素。该函数的原型为

```
void * select(void * a,int size,int p,int r,int i,int( * comp)(void * ,void * ))
```

其中，参数 a 表示序列，size 表示 a 中元素的存储宽度，p、r 表示 a 的首尾元素下标，i 表示要找的是 a 的第 i 序统计，comp 表示 a 中元素的比较规则。

2. 逆序问题

我们看到，比较型排序过程通过比较前后两个元素 $A[i]$、$A[j]$($i<j$)的大小，来决定元素的位置是否需要调整：在升序排序过程中，若 $A[i]>A[j]$，则意味着这两个元素的相对位置需要发生改变。对 $i<j$ 时有 $A[i]>A[j]$，称 i、j 构成 A 的一个逆序。逆序的概念出现在很多学科中，例如下列的 Get the Inversion 问题就需要把计算整数序列的逆序数问题与比较型排序联系起来思考。

Get the Inversion

Description

This term Amy begins to learn another important Course—Linear Algebra. When she comes to the determinant, she finds it boring to calculate the inversion of a given sequence every time. So she asked for her best friend Ray to write a Program to solve it. She would be his partner when dancing Rumba in return. The inversion is defined as follows: for a given sequence A, the inversion is the total number of pairs $<i,j>$ satisfies $i<j$ and $A[i]>A[j]$.

Input

The input file contains multiple test cases. The first line contains the size of the sequence, an integer $N(1\leqslant N\leqslant 500000)$. Then N integers come indicating the element of the sequence. The element is an integer no more than 1,000,000,000. $N=0$ means the

end of the input and should not be processed.

Output

Output exactly one integer for each test case: the inversion of the given sequence.

Sample Input

```
3
1 2 3
2
2 1
0
```

Sample Output

```
0
1
```

(题目来源:ACM/ICPC 2007 南京航空航天大学)

1. 问题描述与分析

Amy 在学习行列式时,常为计算由自然数 $1,2,\cdots,n$ 构成的序列的逆序数而感到烦恼。允诺以晚会舞伴作为回报,请好朋友 Ray 为她编写一个用来计算序列的逆序数程序。

一般地,设 $A[1..n]$是一个具有 n 个不同元素的数组。若 $i<j$ 且 $A[i]>A[j]$,则数偶(i,j)称为 A 的一个**逆序**。例如,序列 $A=\langle 2,3,8,6,1\rangle$中,$\langle 1,5\rangle$、$\langle 2,5\rangle$、$\langle 3,5\rangle$、$\langle 4,5\rangle$、$\langle 3,4\rangle$是 5 个逆序,这是因为 $A[1]=2>1=A[5]$、$A[2]=3>1=A[5]$、$A[3]=8>1=A[5]$、$A[4]=6>1=A[5]$、$A[3]=8>6=A[4]$。本题就是要计算给定数组 $A[1..n]$中逆序个数。问题形式化如下。

输入:数组 $A=\langle A[1],A[2],\cdots,A[n]\rangle$。

输出:A 的逆序个数。

我们知道,序列的逆序数与比较型排序有着密切的关系:逆序意味着元素存储位置的改变。利用这一观察,能够用 $\Theta(n\lg n)$的时间把序列的逆序数计算出来。定义一个全局变量 *count*。在调用过程 MERGE 将两个有序子序列 $A[p..q]$和 $A[q+1..r]$合并成一个有序序列 $A[p..r]$时,要重复比较 $L[i]$与 $R[j]$。当 $L[i]> R[j]$时,就发生 n_1-i+1 个逆序,这是因为 $L[1..n_1]$是有序的,$L[i]> R[j]$时,必有 $L[n_1]\geqslant L[n_1-1]\geqslant\cdots\geqslant L[i]>R[j]$,也就是说 $L[n_1]$、$L[n_1-1]$、$\cdots$、$L[i]$相对于 $R[j]$构成 n_1-i+1 个逆序。把它记录下来。在对一个序列做归并排序的过程中,将利用 *count* 计算出该序列的逆序数。

2. 算法描述

INVERSE-MERGE(A,p,q,r)

$n_1 \leftarrow q-p+1$

$n_2 \leftarrow r-q$

创建数组 $L[1..n_1]$和 $R[1..n_2]$

将 $A[p..q]$复制到 $L[1..n_1]$

将 $A[q+1..r]$复制到 $R[1..n_2]$

```
i←1,j←1
k←p
while i<n₁ and j<n₂
    do if L[i]≤R[j]
        then A[k]←L[i]
             i←i+1
        else A[k]←R[j]
             j←j+1
             count←count+n₁−i+1          ▷记录 n₁−i+1 个逆序
if i<n₁
    then 将 L[i..n₁]复制到 A[k..r]
if j<n₂
    then 将 R[i.. n₂]复制到 A[k..r]

GET-THE-INVERS(A,p,r)
if p<r
   then q←⌊(p+r)/2⌋
        GET-THE-INVERS (A,p,q)
        GET-THE-INVERS (A,q+1,r)
        INVERSE-MERGE(A,p,q,r)
```

算法 3-11　改造 MERGE 和 MERGE-SORT 过程，解决 Get the Inversion 问题的伪代码过程

显然，由于这个算法是基于归并排序的，所以它的时间复杂度是 $\Theta(n\lg n)$。因此，这会让 Amy 很开心。

3. 程序实现

算法 3-11 的 C 程序实现与 3.1.1 节的程序 3-1 和 3.2.1 节的程序 3-6 十分相近，读者可打开 chap03/Get The Inversion 文件夹内的源文件 Get The Inversion.c 研读。

3.4　堆与基于堆的优先队列

本节讨论一个非常重要的数据结构——二叉堆以及基于堆的优先队列。

3.4.1　堆的概念及其创建

1. 二叉堆的概念

数据结构**二叉堆**简称为**堆**，是一个数组对象 A，它可被视为一棵几乎完全的二叉树[①]。树中的每一个结点对应于数组中的一个元素，该元素存储了结点的值。其中 $A[1]$为树根，

① 完全二叉树是具有下述性质的二叉树：树中所有叶子结点的深度一致，且每一个内点都有两个孩子。

设 $A[i]$为树中一个内点,则其左孩子为 $A[2i]$;若 $A[i]$有右孩子,则其右孩子为 $A[2i+1]$。该树除了最底层,完全填满了,最底层的填充是从左到右进行的。表示堆的数组 A 具有两个属性:$length[A]$,它表示数组中的元素个数,以及 $heap\text{-}size[A]$,它表示存储于 A 中的堆的元素个数,它们之间具有 $heap\text{-}size[A] \leqslant length[A]$ 关系。树根 $A[1]$没有父亲结点,对给定结点的下标 $i(>1)$,其父亲的下标 PARENT(i),左孩子的下标 LEFT(i),右孩子的下标 RIGHT(i)可以直接算得:

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2i
RIGHT(i)
    return 2i+1
```

算法 3-12　存储在数组中的二叉树父子结点间下标换算算法

将一个最大堆视为图 3-9(a)所示的一棵二叉树和图 3-9(b)所示的一个数组。树中每个结点圈中的数是存储于该结点的值。结点上方的数对应于它在数组中的下标。数组上方与下方的线条表示父-子关系;父亲总是位于孩子的左边。树的高度为 3;下标为 4 的结点(值为 8)的高度为 1。

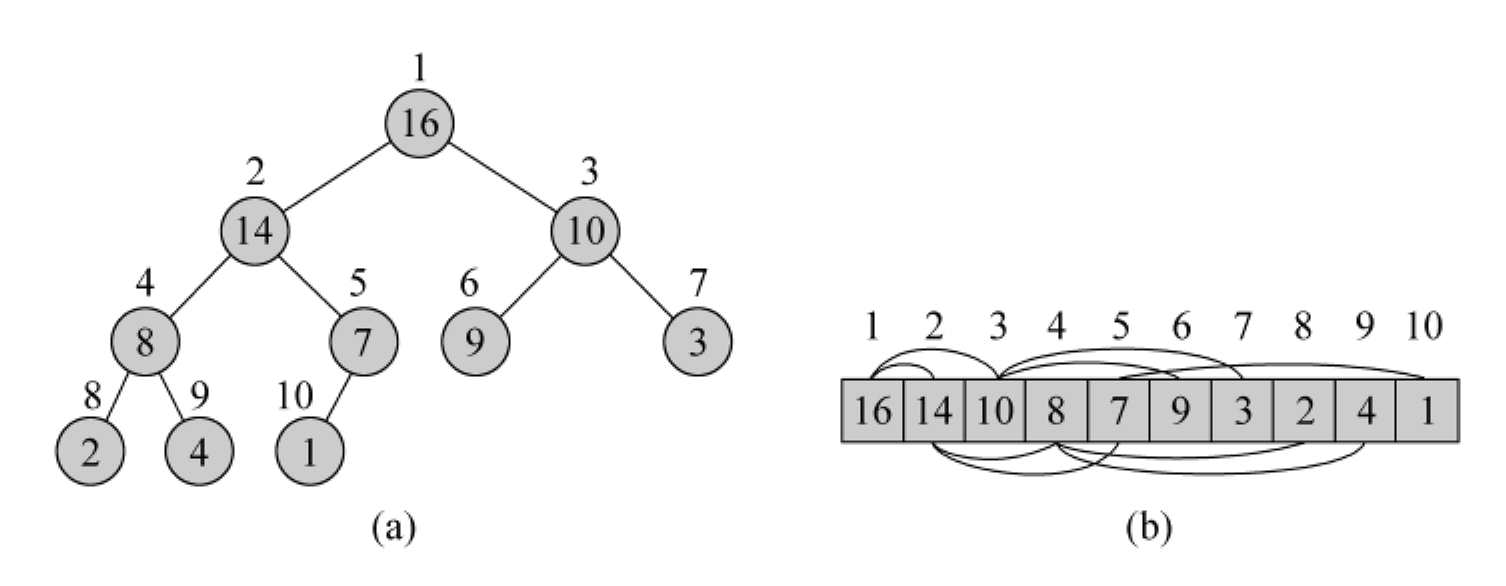


图 3-9　二叉堆

堆有两种:**最大堆**和**最小堆**。在最大堆中,**最大堆的性质**是对每一个非根结点 i,$A[\text{PARENT}(i)] \geqslant A[i]$,即结点的值至多为其父亲的值。于是,最大堆中最大的元素就存储在根中,而以某结点为根的子树中所含的所有结点值不会大于该结点的值。最小堆是以相反的形式组织的;**最小堆的性质**是对每一个非根结点 i,$A[\text{PARENT}(i)] \leqslant A[i]$。最小堆中的最小元素就是根。

特殊地,若几乎完全二叉树仅含一个元素(也是树根),则可自然地视为一个堆,称其为**单元素堆**。

关于存储于数组中的几乎满二叉树,可以不加证明地罗列下面两个有用的结论。

定理 3-4　具有 n 个元素的数组表示的几乎满二叉树的高度 h 满足 $h=\Theta(\lg n)$。

定理 3-5　在存储于数组 $A[1..n]$中的几乎满二叉树,$A[\lfloor n/2 \rfloor+1]$、$A[\lfloor n/2 \rfloor+2]$、…、$A[n]$都是叶子。

2. 二叉堆的性质维护

1）问题描述

本节以下部分以最大堆为例，讨论堆的创建。首先，需要能够维护堆的性质。所谓堆性质的维护，是假定在数组 A 中，元素 $A[i]$为根的完全二叉树的左、右两个孩子都已构成堆，但 $A[i]$可能与它的两个孩子相比不符合堆性质，需要调整 $A[i]$与其子孙的位置使得 $A[i]$为根的完全二叉树成为一个堆。该问题的形式化表示如下。

输入：数组 $A[1..n]$存储一棵几乎完全二叉树，正整数 i。其中以 $A[i]$为根的子树中，$A[i]$的两个孩子已构成最大堆。

输出：数组 A。其中，以 $A[i]$为根的子树中的结点布局有所变动构成最大堆。

解决此问题的想法：比较 $A[i]$、$A[i]$的左孩子 $A[l]$、右孩子 $A[r]$的大小，取最大者与 $A[i]$交换位置。这样原问题就转换成了左子树或右子树的树根的堆性质维护问题，递归地解决子问题。

图 3-10 所示为 MAX-HEAPIFY(A,2)的作用，其中 $heap\text{-}size[A]=10$。图 3-10(a)为初始格局，在结点 $i=2$ 处 $A[2]$违背了最大堆性质，因为它没有它的孩子大。对结点 2 的最大堆性质的恢复是通过在图 3-10(b)中交换 $A[2]$和 $A[4]$，而这又破坏了结点 4 的最大堆性质。递归调用 MAX-HEAPIFY(A,4)，现在的 i 为 4。交换 $A[4]$和 $A[9]$后，如图 3-10(c)所示，结点 4 已修正，递归调用 MAX-HEAPIFY(A,9)时的数据结构不再变化。

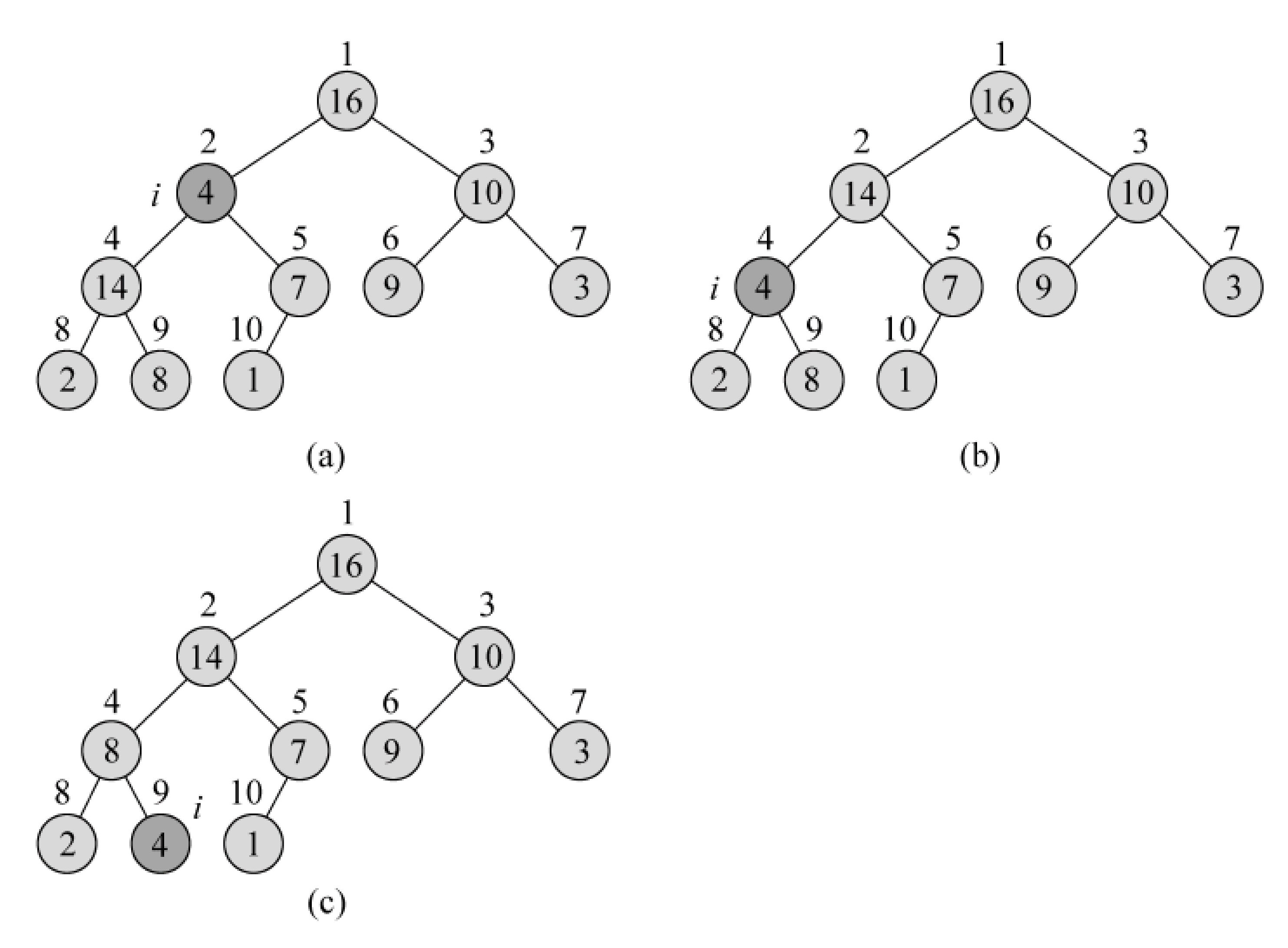


图 3-10　MA-HEAPIFY(A,2)的作用

2）算法的伪代码描述

将算法思想写成伪代码，如下所示。

```
MAX-HEAPIFY(A,i)
1 l←LEFT(i)
2 r←RIGHT(i)
3 if l≤heap-size[A] and A[l]>A[i]
```

```
4      then largest←l
5      else largest←i
6 if r≤heap-size[A] and A[r]>A[largest]
7      then largest←r
8 if largest≠i
9      then exchange A[i]↔A[largest]
10          MAX-HEAPIFY(A,largest)
```

算法 3-13　最大堆性质维护算法

3）算法的运行时间

下面来考虑 MAX-HEAPIFY 过程在最坏情形下的运行时间 $T(n)$，其中 n 表示以 $A[i]$ 为根的子树的结点数。注意在过程中除了第 10 行对自身的递归调用外，所做的所有操作都在常数时间内完成。所以，最坏情形取决于递归调用时处理的子问题的规模。什么情况下子问题的规模最大呢？注意到堆是一棵几乎完全的二叉树，所以，最坏情形发生在子问题是 $A[i]$的左孩子是满二叉树且恰比右孩子多一层。设 $A[i]$为根的子树高为 h，则左孩子高为 $h-1$，而右孩子的高为 $h-2$(见图 3-11)。

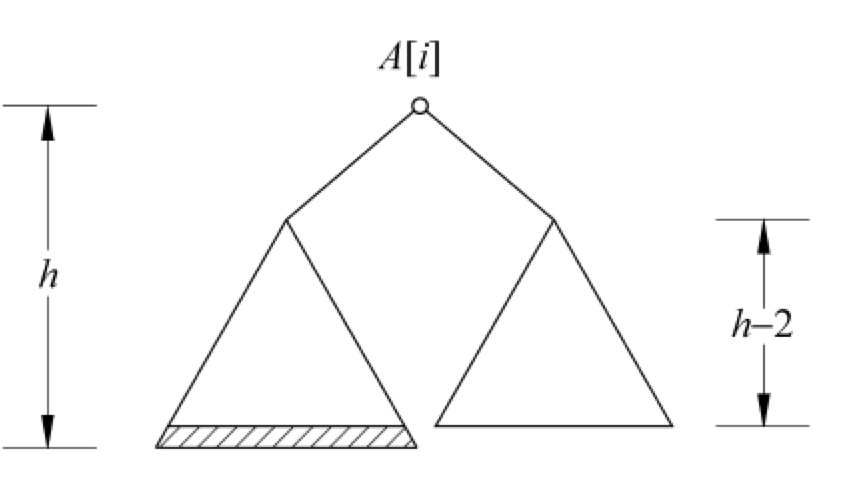


图 3-11　A[i]的子树的规模

右子树的规模为 $2^{h-2}-1$，左子树的规模为右子树的规模 $2^{h-2}-1$ 加上最后一层叶子数(图 3-11 中带有阴影部分)，而这些叶子共有 2^{h-2} 片。因此，$A[i]$为根的子树的结点数 $n=3\cdot 2^{h-2}-3$，左子树规模为 $2\cdot 2^{h-2}-1\approx 2n/3$。于是，得到 MAX-HEAPIFY 过程在最坏情形下的运行时间的递归方程为 $T(n)=T(2n/3)+\Theta(1)$，其中 $\Theta(1)$表示常数时间。令 $a=1, c=3/2$，利用定理 3-1，不难解出 $T(n)=\Theta(\lg n)$。利用定理 3-3，上述结论还可以描述为 MAX-HEAPIFY 过程在最坏情形下的运行时间渐近为以$A[i]$为根的子树的高度 h。

由于算法 3-13 属于末尾递归，故很容易将其转换成如下等价的迭代版本。

```
MAX-HEAPIFY(A,i)
1 l←LEFT(i)
2 r←RIGHT(i)
3 if l≤heap-size[A] and A[l]>A[i]
4      then largest←l
5      else largest←i
6 if r≤heap-size[A] and A[r]>A[largest]
7      then largest←r
8 while largest≠i
9      do exchange A[i]↔A[largest]
10        i←largest
11        l←LEFT(i)
12        r←RIGHT(i)
13        if l≤heap-size[A] and A[l]>A[i]
14            then largest←l
```

```
15      else largest←i
16    if r≤heap-size[A] and A[r]>A[largest]
17      then largest←r
```

算法 3-14　维护最大堆性质过程的迭代版本

算法过程的名称仍然沿用 MAX-HEAPIFY，前 7 行的操作与递归版本中的一致，计算 $A[i]$的左、右孩子下标 l、r。并计算 $A[i]$、$A[l]$、$A[r]$中最大者下标 *largest*。第 8～17 行的 **while** 循环替代了算法 3-13 中的递归操作。循环条件是 $largest \neq i$，即 $A[i]$不能成为一个最大堆的根。循环体中第 9 行执行交换 $A[i]$与 $A[largest]$，第 10～12 行调整下标 i、l、r，第 13～17 行重新计算 $A[i]$、$A[l]$、$A[r]$中最大者下标 *largest*。

3. 二叉堆的创建

1）问题的描述

接下来解决堆的创建问题。

输入：数组 $A[1..n]$。

输出：重排后的数组 $A[1..n]$，元素间构成一个堆。

2）算法的伪代码描述

利用过程 MAX-HEAPIFY 以自底向上的方式将数组 $A[1..n]$转换为一个最大堆。由于子数组 $A[\lfloor n/2 \rfloor+1..n]$的每一个元素都没有左、右孩子，所以都是树的叶子结点，因此每一个均构成为一个单元素堆。过程 BUILD-MAX-HEAP 检测树中的其余结点并对每个结点运行 MAX-HEAPIFY。

```
BUILD-MAX-HEAP(A)
1 heap-size[A]←length[A]
2 for i←⌊length[A]/2⌋downto 1
3    do MAX-HEAPIFY(A,i)
```

算法 3-15　创建最大堆算法

图 3-12 所示为 BUILD-MAX-HEAP 的操作。图 3-12(a)为 10 个元素的输入数组 A 以及由它表示的二叉树。图形展示了调用 MAX-HEAPIFY(A,i)前，循环下标 i 指向结点 5。图 3-12(b)为结果数据结构。下一次重复的循环下标 i 指向结点 4。图 3-12(c)～图 3-12(e)为 BUILD-MAX-HEAP 的 **for** 循环后来的各次重复。请注意无论 MAX-HEAPIFY 对哪个结点调用，该结点的两棵子树都已是最大堆。图 3-12(f)为 BUILD-MAX-HEAP 完成后的最大堆。

3）算法的正确性

BUILD-MAX-HEAP 是一个典型的渐增型过程。设 n 为 $length[A]$，可以归纳出如下循环不变量。

在第 2 行和第 3 行的 **for** 循环每次重复之初，每个结点 $A[i+1]$、$A[i+2]$、…、$A[n]$都是一个最大堆的根。

先用归纳法证明这一循环不变量。

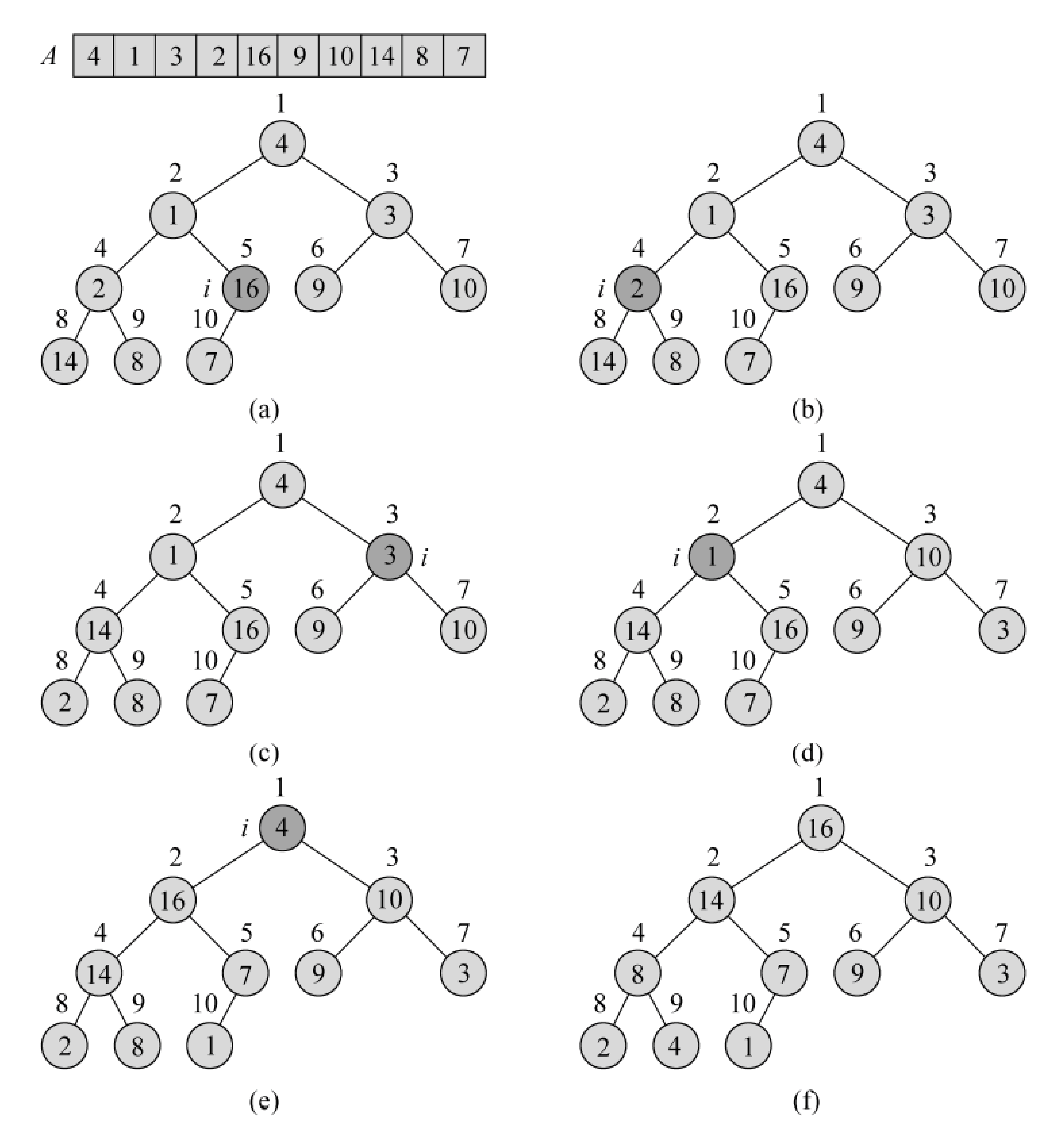


图 3-12 BUILD-MAX-HEAP 的操作

对选环的重复次数 j 而言，$j=1$ 时，$i=\lfloor n/2 \rfloor$。此时，根据定理 3-4 知，$A[\lfloor n/2 \rfloor+1]$，$A[\lfloor n/2 \rfloor+2]$，…，$A[n]$都是叶子，故满足循环不变量。

设 $1<j<\lfloor n/2 \rfloor$，对应的$\lfloor n/2 \rfloor> i>1$。假定此时满足循环不变量为真，即 $A[i+1]$，$A[i+2]$，…，$A[n]$都是一个最大堆的根。在本次重复中，第 3 行调用过程 MAX-HEAPIFY (A,i)，使得 $A[i]$，$A[i+1]$，$A[i+2]$，…，$A[n]$都是最大堆的根。下一次重复前 i 减小 1，这就表述为 $A[i+1]$，$A[i+2]$，…，$A[n]$都是最大堆的根。至此，循环不变量得到证明。

当循环结束时，$i=0$，利用此循环不变量，$A[1]$，$A[2]$，…，$A[n]$均为最大堆的根。

4）**算法的运行时间**

假定序列 A 中有 n 个元素。BUILD-MAX-HEAP 过程的主体是第 2 行和第 3 行的 **for** 循环。循环重复 $n/2$ 次，每次重复调用 MAX-HEAPIFY 过程，耗时 $O(\lg n)$。所以它的时间复杂度为 $O(n\lg n)$。利用其几乎完全二叉树的特性，可以更精细地计算出 BUILD-MAX-HEAP 过程的运行时间为 $\Theta(n)$。

注意算法 BUILD-MAX-HEAP 的第 3 行调用过程 MAX-HEAPIFY(A,i)调整以 $A[i]$为根的子树的堆性质耗时 $\Theta(h)$，$0\leqslant h\leqslant \lg n$。可以用归纳法证明，在一棵具有 n 个结点的几乎完全二叉树中，高度为 h 的子树有 $n/2^{h+1}$棵。于是，算法 3-15 中第 2 行和第 3 行的 **for** 循环耗时可表示为

$$\begin{aligned}
\sum_{h=0}^{\lg n} h \cdot \frac{n}{2^{h+1}} &= \frac{n}{2}\sum_{h=0}^{\lg n} h\left(\frac{1}{2}\right)^{h} = \frac{n}{4}\sum_{h=0}^{\lg n} h\left(\frac{1}{2}\right)^{h-1} \\
&\overset{x=1/2}{=} \frac{n}{4}\sum_{h=0}^{\lg n} hx^{h-1} < \frac{n}{4}\sum_{h=0}^{\infty} hx^{h-1} \quad \text{（正项级数有限项之和小于无限项之和）} \\
&= \frac{n}{4}\sum_{h=0}^{\infty}(x^{h})' = \frac{n}{4}\left(\sum_{h=0}^{\infty}x^{h}\right)' \quad \text{（幂级数在收敛域内可逐项求导）} \\
&= \frac{n}{4}\left(\frac{1}{1-x}\right)' = \frac{n}{4}\cdot\frac{1}{(1-x)^2} \quad \left(\text{幂级数}\sum_{h=0}^{\infty}x^{h}\text{ 在收敛域中的和函数为}\frac{1}{1-x}\right) \\
&\overset{x=1/2}{=} \frac{n}{4}\cdot 4 = n
\end{aligned}$$

有了 BUILD-MAX-HEAP 过程后，就可以在 $\Theta(n)$时间内将一个数组创建为一个堆。

4. 程序实现

将实现算法 3-12、算法 3-14 和算法 3-15 的 C 函数原型声明如下。

```
1 int left(int i);
2 int right(int i);
3 int parent(int i);
4 void heapify(void *a,int size,int i,int heapSize,int(*comp)(void *,void *));
5 void buildHeap(void *a,int size,int length,int(*comp)(void *,void *));
```

程序 3-14　堆操作函数的原型声明

其中，第 1～3 行将实现算法 3-12 的 3 个父子转换过程，第 4 行将实现算法 3-14 的堆性质维护过程，第 5 行将实现算法 3-15 的堆创建过程。把这些函数原型声明代码存储为 utility 文件夹中的头文件 heap.h。

1）父子结点换算函数

```
int left(int i){
    return 2*i+1;
}
int right(int i){
    return 2*i+2;
}
int parent(int i){
    return (i-1)/2;
}
```

程序 3-15　实现算法 3-12 的 C 函数

函数 left、right 和 parent 分别实现算法 3-12 中的 LEFT、RIGHT 和 PARENT 过程。与对应伪代码过程相比，参数是一致的，都是表示结点在数组中的下标 i。返回值的表达式与伪代码中的有所不同，原因是伪代码中数组的下标是从 1 开始的，而 C 语言中的数组下标是从 0 开始的。

2）**堆性质维护函数**

此处实现维护堆性质的迭代过程。

```
void heapify(void *a,int size,int i,int heapSize,int(*comp)(void const *,void const *)){
    int l=left(i),r=right(i),most;
    if(l<heapSize&&comp((char *)a+l*size,(char *)a+i*size)>0)
        most=l;
    else
        most=i;
    if(r<heapSize&&comp((char *)a+r*size,(char *)a+most*size)>0)
        most=r;
    while(most!=i){
      swap((char *)a+i*size,(char *)a+most*size,size);
      i=most;
      l=left(i),r=right(i);
      if(l<heapSize&&comp((char *)a+l*size,(char *)a+i*size)>0)
        most=l;
      else
        most=i;
      if(r<heapSize&&comp((char *)a+r*size,(char *)a+most*size)>0)
        most=r;
    }
}
```

程序 3-16　实现算法 3-14 的 C 函数

对程序 3-16 的说明如下。

(1)由于目标是开发出适用于最大堆和最小堆,堆中元素可以是任何类型的数据的通用过程,所以除了要向函数传递作为问题输入的数组 A 和子树根结点位置 i 以外,还需要传数组元素的存储宽度 size,元素间的比较规则 comp。因为堆性质将维持在序列 A 中,所以无须返回任何值。此外,上文提到,表示堆的数组 A 具有两个域:*length*[A],它表示数组中的元素个数;以及 *heap-size*[A],它表示堆中的元素个数。由于仅仅将一个数组作为堆空间,并未将其作为一个独立的数据类型,所以可以将这些属性作为参数加以传递,在本函数中需要传递表示堆中元素个数的 heapSize,用来对结点下标进行合法性检测。

(2)根据算法伪代码的上下文知,程序中需设置两个下标变量 l 和 r 来记录 $A[i]$的左、右孩子的位置。此外,算法是专门用来维护最大堆的,有一个用来跟踪 $A[i]$、$A[l]$、$A[r]$中最大者的变量 *largest*,而所实现的程序既要能用于最大堆也能用于最小堆,所以对这一变量命名为 *most*。

(3)由于用序列中元素的比较规则 comp 来控制堆的类型(最大堆或最小堆),所以在判断元素大小关系处都以 comp 为准。此外,利用程序 3-3 定义的 swap 函数来执行交换两个变量的操作。

3）**堆创建函数**

```
void buildHeap(void *a,int size,int length,int(*comp)(void *,void *)){
      int i;
```

```
    for(i=length/2;i>=0;i--)
        heapify(a,size,i,length,comp);
}
```

程序 3-17　实现算法 3-15 的 C 函数

对程序 3-15 的说明如下。

(1) 算法解决的是最大堆的创建问题,我们的目标是开发一个既能创建最大堆又能创建最小堆的程序过程,所以传递给过程的参数除了数组 A 以外,还需传递数组元素的存储宽度 size、数组长度 length 以及元素间的比较规则 comp。创建的堆维持在序列 A 的存储空间内,所以无须返回任何值。

(2) 程序中需要设置一个整型的循环控制变量 i。由于已经知道了将要创建的堆空间的长度就是堆长度,所以无须再对堆长度做任何修改。

将程序 3-15 至程序 3-17 的代码保存在 utility 文件夹中的源文件 heap. c 中,以备重用。

3.4.2　基于二叉堆的优先队列

优先队列就是进入队列的每一个元素都有各自的优先级,每次出队操作的对象是队列中优先级最高的元素。往往用一个线性表来实现优先队列,实现的方案有多种。例如,每次出队操作前,对线性表按优先级排序,然后将优先级最高的元素(此时,该元素必在表首或表尾)出队。如果排序过程采用比较型算法,则出队操作至少耗时 $\Theta(n\lg n)$(见 3.3.1 节)。还可以利用选择算法 SELECT(见 3.2.3 节),选择表中优先级最高的元素出队,这也至少要消耗 $\Theta(n)$时间。

我们知道,存储堆的数组的第一个元素就是最大的(或最小的),所以可以利用堆来作为优先队列的元素载体。

优先队列有两个基本操作:入队操作 ENQUEUE(Q,e),其中参数 Q 是优先队列,e 是要加入队列的元素。出队操作 DEQUEUE(Q),它将返回 Q 中优先级最大(最小)。下面来讨论利用最大堆的最大优先队列 Q,假定 Q 是一个最大堆。

1. 入队算法描述与分析

对于入队操作,把要加入的元素放在堆的末尾,然后维护堆性质——从新的末尾开始,检测到当前元素的优先级大于其父亲结点元素的优先级,两者交换。

```
ENQUEUE(Q,e)
if heap-size[Q]=length[Q]
  then error "堆上溢"
i←heap-size[Q]←heap-size[Q]+1
A[i]←e
while i>1 and A[PARENT(i)]<A[i]
  do exchange A[i]↔A[PARENT(i)]
     i←PARENT(i)
```

算法 3-16　基于最大堆的优先队列入队算法

显然,ENQUEUE 过程的运行时间取决于第 5～7 行的 **while** 循环的重复次数。循环体中的第 7 行是将当前结点变换为父结点,所以该循环至多重复堆的树高 $\lg n$ 次。所以,ENQUEUE 过程的运行时间为 $\Theta(\lg n)$。

2. 出队算法描述与分析

对于出队操作,只要将堆 Q 中的最大元素($Q[1]$)"舍弃":$Q[1]$ 的值暂存于 max,将 $Q[heap\text{-}size[Q]]$赋予 $Q[1]$,并使 $heap\text{-}size$ 减小 1。然后维护剩余元素的堆性质,并返回 max。伪代码如下。

```
DEQUEUE (Q)
1 if heap-size[Q]<1
2   then error "堆下溢"
3 max←Q[1]
4 Q[1]←Q[heap-size[Q]]
5 heap-size[Q]←heap-size[Q]-1
6 MAX-HEAPIFY(Q,1)
7 return max
```

算法 3-17 基于最大堆的优先队列出队算法

本过程中除了第 6 行调用 MAX-HEAPIFY 过程耗时 $\Theta(\lg n)$外,其余所有操作都在常数时间内完成,所以出队操作的运行时间为 $\Theta(\lg n)$。

3. 程序实现

作为重要的数据结构,在 C 语言中把基于二叉堆的优先队列定义成如下结构体。

1) **数据类型**

```
1 typedef struct {
2   void *heap;                        /*指向存储队列元素的数组的首元素指针*/
3   int eleSize;                       /*元素存储宽度*/
4   int length;                        /*数组长度*/
5   int heapSize;                      /*堆中的元素个数*/
6   int(*compare)(void *,void *);      /*元素比较函数*/
7 }PQueue;                             /*基于堆的优先队列类型*/
8 PQueue *initPQueue(int size,int n,int(*comp)(void *,void *));    /*创建队列*/
9 void pQueueClr(PQueue *q);           /*清理队列存储空间*/
10 int empty(PQueue *q);               /*检测队列是否为空*/
11 void enQueue(PQueue *q,void *e);    /*入队操作*/
12 void * deQueue(PQueue *q);          /*出队操作*/
```

程序 3-18 定义基于堆的优先队列数据类型并声明优先队列操作函数的 C 代码

对程序 3-18 的说明如下。

(1) 第 1～7 行将优先队列定义为结构体类型 PQueue。它有 5 个属性,heap 是指向存储队列元素的数组的首元素指针,该数组中的数据将被组织成一个堆。eleSize 表示数组中

的元素存储宽度。length 表示数组的长度。heapSize 表示堆中的元素个数。compare 表示队列中元素之间优先级大小比较规则。

(2) 第 8 行声明的函数 initPQueue 用来创建一个容量为 n、元素宽度为 size、优先级大小比较规则为 comp 的优先队列。第 9 行声明的函数 pQueueClr 用来对程序中不再使用的优先队列 q 清理内存空间，之所以要这样做是因为队列中 heap 指针指向一块动态分配的内存，废弃前应将这块内存释放掉。第 10 行声明的函数 empty 用来检测优先队列 q 是否为空。这 3 个函数对优先队列做常规维护操作。

(3) 第 11 行和第 12 行声明的函数 enQueue、deQueue 分别实现算法 3-15 和算法 3-16 的 ENQUEUE 和 DEQUEUE 过程。

为便于代码重用，将程序 3-18 存储为文件夹 datastructure 中的头文件 pqueue.h。

2) **常规维护操作**

优先队列常规维护函数定义如下。

```
1 PQueue *initPQueue(int size,int n,int(*comp)(void *,void *)){    /*创建空队列*/
2     PQueue *q=(PQueue*)malloc(sizeof(PQueue));
3     assert(q);
4     q->eleSize=size;                                               /*设置元素存储宽度*/
5     q->length=n;                                                   /*设置堆的最大长度*/
6     q->heap=(void *)malloc(n*size);                                /*分配堆空间*/
7     q->heapSize=0;                                                 /*目前队空*/
8     q->compare=comp;                                               /*设置元素比较规则*/
9     return q;
10 }
11 void pQueueClr(PQueue *q){                                        /*清理队列存储空间*/
12     free(q->heap);
13     q->heap=q->compare=NULL;
14     q->heapSize=q->length=q->eleSize=0;
15 }
16 int empty(PQueue *q){
17     return q->heapSize<1;
18 }
```

程序 3-19　优先队列常规维护操作函数的定义

对程序 3-19 的说明如下。

(1) 第 1～10 行定义的函数 initPQueue 用参数 size 确定队列 q 中元素的存储宽度属性 eleSize(第 4 行)，用参数 n 确定队列 q 最多可存储的元素个数 length(第 5 行)，并为 q 的堆分配的长度为 length，每个元素宽度为 eleSize 的数组空间(第 6 行)。第 7 行将 q 的 heapSize 属性置为 0，表示所创建的队列初始时是空的。第 8 行用参数 comp 设置队列 q 的元素优先级的比较规则。

(2) 第 11～15 行定义的函数 pQueueClr 负责在程序废弃优先队列 q 之前释放其堆空间 heap(第 12 行)，并将其中的各指针属性指向安全的空地址 NULL。

(3) 第 16～18 行定义的函数 empty 通过检测优先队列的堆长度属性 heapSize 是否小

于 0 来判断 q 是否为空。

3) **入队操作**

优先队列的入队操作算法实现如下。

```
1 void enQueue(PQueue *q,void *e){
2     int i,heapSize=q->heapSize,length=q->length,eleSize=q->eleSize;
3     int(*comp)(void *,void *)=q->compare;
4     void* heap=q->heap;
5     assert(heapSize<length);                  /* 防止队列满 */
6     i=heapSize++; q->heapSize++;              /* i 为扩大后的堆的最后元素的下标 */
7     memcpy((char *)heap+i*eleSize,(char *)e,eleSize);          /* heap[i]←e */
8     while (i>0&&comp((char *)heap+parent(i)*eleSize,(char *)heap+i*eleSize)<0){
9         swap((char *)heap+i*eleSize,(char *)heap+parent(i)*eleSize,eleSize);
10        i=parent(i);
11    }
12 }
```

程序 3-20 实现算法 3-16 的入队过程 ENQUEUE 的 C 函数定义

对程序 3-20 的说明如下。

(1) 由于将队列中元素的存储宽度和元素间优先级比较规则作为优先队列的属性,所以入队操作函数的参数与伪代码过程的参数一样简洁,参数为队列 q 和要入队的元素 e。

(2) 为使代码简短,第 2～4 行用 q 的各属性初始化相应的简单变量。

(3) 第 6 行的赋值操作 i=heapSize++,i 得到的是 heapSize 自增前的值,heapSize 自增后 heap[i]就是 heap[heapSize-1],即扩展后堆空间最后一个元素。该行中的 q->heapSize++实际完成堆空间的扩展。

4) **出队操作**

优先队列的出队操作算法实现如下。

```
void * deQueue(PQueue *q){
    int heapSize=q->heapSize,length=q->length,eleSize=q->eleSize;
    int(*comp)(void *,void *)=q->compare;
    void* heap=q->heap,*top;
    assert(!empty(q));                          /* 防止队列空 */
    assert(top=(void *)malloc(eleSize));
    memcpy(top,heap,eleSize);                   /* top←heap[0] */
    heapSize--;q->heapSize--;
    memcpy((char *)heap,(char *)heap+heapSize*eleSize,eleSize);
    heapify(heap,eleSize,0,heapSize,comp);
    return top;
}
```

程序 3-21 实现算法 3-17 中出队操作过程 DEQUEUE 的 C 函数

与程序 3-20 相仿,函数 deQueue 的参数非常简洁——只有优先队列 q。操作成功将返回原队列中优先级最高的元素指针。注意,将伪代码中的赋值操作代之以 memcpy 的调用,

比较运算代之以 comp 的调用，函数 deQueue 的代码与过程 DEQUEUE 的伪代码十分相近。

为便于代码重用，将程序 3-19 至程序 3-21 的代码存储为文件夹 datastructure 中的源文件 pqueue. c。

3.4.3　应用

优先队列在现实模拟中有广泛应用，考虑下面的问题。

Department

The Department of Security has a new headquarters building. The building has several floors, and on each floor there are rooms numbered *xxyy* where *yy* stands for the room number and *xx* for the floor number, $0 < xx, yy \leqslant 10$. The building has 'paternoster' elevator, i. e. elevator build up from several cabins running all around. From time to time the agents must visit the headquarters. During their visit they want to visit several rooms and in each room they want to stay for some time. Due to the security reasons, there can be only one agent in the same room at the same time. The same rule applies to the elevators. The visits are planned in the way ensuring they can be accomplished within one day. Each agent visits the headquarters at most once a day.

Each agent enters the building at the 1st floor, passes the reception and then starts to visit the rooms according to his/her list. Agents always visit the rooms by the increasing room numbers. The agents form a linear hierarchy according to which they have assigned their one letter personal codes. The agents with higher seniority have lexico-graphically smaller codes. No two agents have the same code.

If more than one agent wants to enter a room, or an elevator, the agents have to form a queue. In each queue, they always stand according to their codes. The higher the seniority of the agent, the closer to the top of the queue he stands. Every 5 s (seconds) the first agent in the queue in front of the elevator enters the elevator. After visiting the last room in the headquarters each agent uses if necessary elevator to the first floor and exits the building.

The times necessary to move from a certain point in the headquarters to another are set as follows: Entering the building, i. e. passing the reception and reaching the elevator, or a room on the first floor takes 30 s. Exiting the building, i. e. stepping out of the elevator or a room on the first floor and passing the reception takes also 30 s. On the same floor, the transfer from the elevator to the room (or to the queue in front of the room), or from the room to the elevator (or to the queue in front of the elevator), or from one room to another (or to the queue in front of the room) takes 10 s. The transfer from one floor to the next floor above or below in an elevator takes 30s. Write a program that determines time course of agent's visits in the headquarters.

Input

The input contains the descriptions of $n \geqslant 0$ visits of different agents. The first line of the description of each visit consists of agent's one character code C, $C=A, \cdots, Z$, and the time when the agent enters the headquarters. The time is in the format HH:MM:SS (hours, minutes, seconds). The next lines (there will be at least one) contain the room number, and the length of time intended to stay in the room, time is in seconds. Each room is in a separate line. The list of rooms is sorted according to the increasing room number. The list of rooms ends by the line containing 0. The list of the descriptions of visits ends by the line containing the character dot.

Output

The output contains detailed records of each agent's visit in the headquarters. For each agent, there will be a block. Blocks are ordered in the order of increasing agent's codes. Blocks are separated by an empty line. After the last block there is an empty line too. The first line of a block contains the code of agent. Next lines contain the starting and ending time (in format HH:MM:SS) and the descriptions of his/her activity. Time data will be separated by one blank character.

Description will be separated from time by one blank character.

Description will have a form Entry, Exit or Message. The Message can be one of the following:

Waiting in elevator queue
Waiting in front of room *RoomNumber*
Transfer from room *RoomNumber* to room *RoomNumber*
Transfer from elevator to room *RoomNumber*
Transfer from *RoomNumber* to elevator
Stay in room *RoomNumber*
Stay in elevator

Sample Input

```
A 10:00:00
0101 100
0110 50
0202 90
0205 50
0
B 10:01:00
0105 100
0201 5
0205 200
0
```

Sample Output

```
A
10:00:00 10:00:30 Entry
10:00:30 10:02:10 Stay in room 0101
10:02:10 10:02:20 Transfer from room 0101 to room 0110
10:02:20 10:03:10 Stay in room 0110
10:03:10 10:03:20 Transfer from room 0110 to elevator
10:03:20 10:03:50 Stay in elevator
10:03:50 10:04:00 Transfer from elevator to room 0202
10:04:00 10:05:30 Stay in room 0202
10:05:30 10:05:40 Transfer from room 0202 to room 0205
10:05:40 10:07:40 Waiting in front of room 0205
10:07:40 10:08:30 Stay in room 0205
10:08:30 10:08:40 Transfer from room 0205 to elevator
10:08:40 10:09:10 Stay in elevator
10:09:10 10:09:40 Exit

B
10:01:00 10:01:30 Entry
10:01:30 10:03:10 Stay in room 0105
10:03:10 10:03:20 Transfer from room 0105 to elevator
10:03:20 10:03:25 Waiting in elevator queue
10:03:25 10:03:55 Stay in elevator
10:03:55 10:04:05 Transfer from elevator to room 0201
10:04:05 10:04:10 Stay in room 0201
10:04:10 10:04:20 Transfer from room 0201 to room 0205
10:04:20 10:07:40 Stay in room 0205
10:07:40 10:07:50 Transfer from room 0205 to elevator
10:07:50 10:08:20 Stay in elevator
10:08:20 10:08:50 Exit
```

1. 问题描述与分析

安全局大楼内有 10 层,每层有 10 个房间。房间编号由 4 位数字构成:$xxyy$。前两位 $0<xx\leqslant 10$ 表示楼层,后两位 $0<yy\leqslant 10$ 表示层内的房间号。例如 0309 表示 3 层 9 号房,而 0107 表示 1 层 7 号房。大楼装备有电梯,每隔 5s 就有一个箱斗可载一位乘客。上下一层,电梯耗时 30s。

安全局向各地派出 26 个特派员,代号分别为 $A\sim Z$。代号越排前,等级越高,也就是说代号为 A 的等级最高,而代号为 Z 的等级最低。特派员需要返回安全局汇报工作或接受指令。一天内可能有若干个特派员返回安全局,他们有各自的造访计划表,表中各个需造访的房间按编号升序排列(存放在输入文件中),列表中的每一项包括需造访的房间编号和需在该房间内停留的时间。

出于保密的要求,电梯内每次只能运载一个人,若有两个人需要造访一个房间,必须一

个完成访问离开后另一个才能进入房间。因此,在每一个房间及每层的电梯前可能出现一个访问者的等待队列,队列中若有多个特派员来到的时间相同,则按它们的等级排列,即等级越高越优先。于是,房间和电梯的等待队列应当是优先队列。

我们的任务是为这一天所有回访安全局大楼的特派员制作一份实际的行程列表。与他们的计划列表相比,行程列表将标识出每一项活动的实际起止时间以及(如果有的话)在各等待队列中的等待时间。

可以为每一层电梯和每一个房间设置一个优先队列来模拟各特派员一天的活动,跟踪他们实际的活动时间来完成制作每个人的行程列表。

2. 算法描述

1) 数据表示

根据题意说明,一个特派员的行程数据是由大楼内的一个个房间排列而成,每个房间用 4 个数字表示所在楼层与房间号,为算法描述的简洁起见,把房间设为具有楼层 *floor* 和房号 *number* 两个属性的 Room 对象。其中,$1 \leqslant floor \leqslant 10$ 和 $1 \leqslant number \leqslant 10$。若将电梯视为特殊的房间,每层只有一个,令其编号为 0,则可将 *number* 的取值范围扩展为 $0 \leqslant number \leqslant 10$。

特派员的行程表是由一系列的事件组成,这些事件归纳在表 3-1 中。在表中还对不同的事件给出不同的类型编号。

表 3-1　行程表

事　　件	类型	持续时间
进入大楼	0	30s
等待电梯	1	待定,初始化为 0s
等待进入房间	2	待定,初始化为 0s
从房间走到另一个房间	3	10s
从电梯走到房间	4	10s
从房间走到电梯	5	10s
在房间内	6	输入数据
在电梯内	7	每上下一层 30s
走出大楼	8	30s

根据行程表的表示要求,每个事件表示为具有 6 个属性的 Event 对象:特派员代码 *code*(*A*～*Z*)、事件类型 *type*(0～8)、发生时间 *begin*、延续时间 *length*、起始地点 *from*、目标地点 *to*。其中,*from* 和 *to* 都是 Room 对象。

每个特派员都有一个由若干个事件组成的行程表,表的长度各不相同,所以组织成链表是合适的,至多有 26 个特派员,所以可组织成一个元素为链表的数组 *schedules*[*A*..*Z*]来管理所有特派员的行程表。

为了模拟各特派员在大楼内的行为,要为每个房间和电梯设置一个等待队列,这些队列

按特派员的优先级决定出队的顺序，所以应该是优先队列。共有 10 层楼，每层楼有 11 个房间（包括一个电梯），设置一个元素为优先队列的二维数组 *queues*[1..10,0..10]。为简化算法过程的参数，可以把这两个数组定义成全局量。

2）**事件的创建与行程表的初始化**

解决本问题的思路是先根据输入文件中特派员的行程数据创建该特派员整个行程的所有事件组成的行程表，将其中的等待事件的等待时间 *length* 初始化为 0。然后根据特派员们行程中事件的相互影响，正确地加以调整，最后输出调整后的行程表。为每个特派员创建行程表是首先需要考虑的。由于有 9 类不同的事件，所以需要有 9 个创建事件并将时间追加到行程表中的过程。

```
ENTRY(code,begin)
WAIT-ELEVATOR(code,begin,floor)
WAIT-FRONT-ROOM(code,begin,room)
ROOM-TO-ROOM(code,begin,from,to)
ELEVATOR-TO-ROOM(code,begin,to)
ROOM-TO-ELEVATOR(code,begin,from)
IN-ROOM(code,begin,length,room)
IN-ELEVATOR(code,begin,floor1,floor2)
EXIT(code,begin,from)
```

分别用来创建进入大楼、等待电梯、等待进入房间、从一个房间走到另一个房间、从一个房间走到电梯、从电梯走到一个房间、在房间内、乘坐电梯和离开大楼事件。

这 9 个过程中，除了 EXIT 过程可能会调用其他过程以外，其余都是独立地生成一个事件并将其初始化后追加到代码为 *code* 的特派员行程链表 *schedules*[*code*]的尾部。为节省篇幅，下面仅列出 ENTRY、WAIT-ELEVATOR 和 EXIT 的伪代码过程。

```
ENTRY(code,begin)
code[e]←code
type[e]←0
begin[e]←begin
length[e]←30
floor[from[e]]←0,number[from[e]]←0
floor[to[e]]←0,number[to[e]]←0
LIST-PUSH-BACK(schedules[code],e)
return begin[e]+length[e]
```

```
WAIT-ELEVATOR(code,begin,floor)
code[e]←code
type[e]←1
begin[e]←begin
if begin MOD 5≡0
      then length[e]←0
      else length[e]←5-(begin MOD 5)
floor[from[e]]←floor[to[e]]←floor
```

```
number[from[e]]←number[to[e]]←0
LIST-PUSH-BACK(schedules[code],e)
tail←prev[nil[schedules[code]]]
ENQUEUE(queues[floor[to],0],tail)
return begin[e]+length[e]

EXIT(code,begin,from)
code[e]←code
type[e]←8
if floor[from]>1            ▷不是从 1 楼离开
   then begin←ROOM-TO-ELEVATOR(code,begin,from)
        begin←WAIT-ELEVATOR(code,begin,1)
        begin←IN-ELEVATOR(code,begin,floor[from],1)
begin[e]←begin
length[e]←30
from[e]←from
floor[to[e]]←0,number[to[e]]←0
LIST-PUSH-BACK(schedules[code],e)
```

算法 3-18 创建各类事件的过程

这些过程的基本工作就是设置事件的各属性初始值,并将事件追加到行程链表中。需要注意如下问题。

(1) 每个过程都返回本事件的完成时间作为创建下一个事件的开始时间的参数 *begin*。决定 ENTRY 过程所创建的进入事件的开始时间的参数 *begin* 是来自输入文件中的数据。

(2) WAIT-ELEVATOR 创建等待电梯事件,由于电梯每隔 5s 就有 1 班,所以乘上电梯的时间应是 5 的倍数。于是开始时间若不是 5 的倍数,则等待时间应初始化为 5 与开始时间除以 5 的余数之差,这样就可以在电梯到达时乘上电梯了。此外,应注意该过程创建的等待电梯事件(以及 WAIT-FRONT-ROOM 创建的等待进入房间事件)加入到行程链表中后,成为链表中的尾结点,需将此结点加入到相应的等待队列中。

(3) EXIT 过程需要考虑离开大楼前是否在 1 楼,如果不是从 1 楼离开的,则需调用 ROOM-TO-ELEVATOR 从房间走到电梯,然后调用 WAIT-ELEVATOR 等待电梯,还需调用 IN-ELEVATOR 乘坐电梯到 1 楼。

利用这些过程可以完成对输入文件中特派员一项行程数据进行预处理的过程。在输入文件中,对一个特定的特派员(由起点信息中的代码 *code* 决定),除了起点信息外,每项行程信息仅包含 2 个数据:目的房间 *to* 和在目的房间中停留的时间 *length*。要处理一项信息还需要 2 个信息:起点地点 *from* 和起始时间 *begin*。这两个信息都来自于上一个加入行程表的事件 *e*:目的地 *to*[*e*]和起始时间与延续时间之和 *begin*[*e*]+*length*[*e*]。用 TO-ROOM 过程来处理特派员 *code* 的一项行程信息,该过程的参数除了代码 *code* 以外,还包含来自于上一项信息处理结果的起点 *from*、开始时间 *begin* 以及本项信息中的延续时间 *length* 和目标地点 *to*。

```
TO-ROOM(code,begin,length,from,to)
```

```
if floor[from]=floor[to]                    ▷同层
    then if(number[from]≠0)                 ▷非来自入口
        then begin←ROOM-TO-ROOM(code,begin,from,to)
else if number[from]≠0
    then begin←ROOM-TO-ELEVATOR(code,begin,from)
        begin←WAIT-ELEVATOR(code,begin,floor[from])
        begin←IN-ELEVATOR(code,begin,floor[from],floor[to])
        begin←ELEVATOR-TO-ROOM(code,begin,to)
        begin←WAIT-FRONT-ROOM(code,begin,to)
return IN-ROOM(code,begin,length,to)
```

算法 3-19　处理一项行程数据的伪代码过程

该过程首先分别对本项行程信息中起点与终点是否在同一层楼进行不同的处理。如果起点与终点在同一层且起点是从同层的另一房间，则调用 ROOM-TO-ROOM 过程，加入一个从房间 *from* 走到房间 *to* 的事件。起点与终点在同层且起点就是大楼入口，由于进入时已计算了走到房间的时间，所以省略 ROOM-TO-ROOM 过程的过程。如果起点与终点不在同层且起点不是大楼入口而是一房间，则需调用 ROOM-TO-ELEVATOR 过程，加入一个从房间走到电梯的事件。起点与终点不在同层且起点就是大楼入口才进入大楼，由于进入时已计算了走到电梯的时间，所以省略 ROOM-TO-ELEVATOR 过程的调用。无论如何，只要起点与终点不在同一层，都需要加入走到电梯、等待电梯和乘坐电梯的事件。

然后，调用 WAIT-FRONT-ROOM 过程，追加一个在终点 *to* 房前等待的事件，最后调用 IN-ROOM 过程，追加一个留在终点 *to* 房中的事件。

利用 ENTRY、TO-ROOM 和 EXIT 过程，可以写出如下的根据输入文件中包含的各特派员的访问行程的数据创建其行程表的算法过程。

```
INIT-SCHEDULES(f)
read code,begin from f
while code≠'.'
  do begin←ENTRY(code,begin)
     from←"0000"
     read to,length from f
     while to ≠'0'
         do begin←TO-ROOM(code,begin,length,from,to)
           from←to
           read to,length from f
      EXIT(code,begin,from)
      read code,begin from f
```

算法 3-20　创建所有特派员行程表的伪代码过程

输入文件含有若干个特派员的行程数据。对每个特派员，行程数据由若干行组成。第一行包括两个信息：代码 *code* 和以 hh:mm:ss 格式表示的起始时间 *begin*。然后，若干行关于该特派员的行程数据，每行包含两个信息，以 *xxyy* 格式表示的访问的房间 *to*，其中 *xx* 表示楼层，*yy* 表示层内的房号。以及访问该房间的持续时间 *length*。以仅含数字 0 的一行

数据作为该特派员行程数据的结束标志。整个文件以仅含字符“.”的一行作为结束标志。根据文件的格式,过程首先从输入文件 f 中读取第一个特派员的代码 *code* 和开始时间 *begin*。然后通过第 2～11 行的 **while** 循环处理每个特派员的行程数据。其中,第 3 行调用 ENTRY 过程向行程表加入进入大楼的事件,返回的值赋予 *begin* 作为下一个事件的开始时间。第 4 行将该次行程的起点的 *from* 置为特殊值 0000,表示刚进入大楼。第 5 行从 f 中读取特派员进楼后的第一项行程的目标房间 *to* 和访问持续时间 *length*。第 6～9 行的 **while** 循环处理该特派员的每一项行程数据。其中,第 7 行调用 TO-ROOM 过程处理该项行程数据,返回的值赋予 *begin*,作为下一项行程的开始时间。第 8 行将本次的目标房间 *to* 赋予 *from*,作为下一项行程的起点房间。第 9 行读取该特派员的下一项行程数据。一旦该特派员的输入数据处理完毕(读到 *to* 为 0),意味着他/她应当离开大楼了,第 10 行调用 EXIT 过程在行程链表中加入离开事件。

3) **行程表的处理**

为了模拟各位特派员在大楼中的活动,需要做如下控制。设置一个工作矩阵 *work-matrix*[1..10,0..10],用来跟踪所有等待队列上一次出队的队首。模拟过程首先选取所有等待队列的队首中的最小者 x,然后通过一个循环逐一对目前 *queues* 的各队首的最小者 x(假定其所在等待队列为 *queues*[i,j]),酌情(与 *work-matrix*[i,j]的相关数据比较,看是否需要等待)修改其持续时间 *length*,并根据对它的修改,逐一修改该事件(从属于某个特派员)所在行程链表中其后的每个事件的开始时间,如果这样的事件处于某等待队列中则需维护该队列的堆性质。然后,将 x 交给 *work-matrix*[i,j]作为下一次对 *queues*[i,j]中出队的队首作比较用。并从 x 所在的等待队列中,将其弹出,这样的操作循环往复,直至 *queues* 中所有队列为空。

```
PROCESS-SCHEDULES()
allocate work-matrix[1..10,0..10] and set work-matrix[i,j] to NIL
x←MINI(queues)
while x≠NIL
  do i←floor[to[key[x]]],j←number[to[key[x]]],flag←false
     if work-matrix[i,j]≠NIL                    ▷x 非等待队列第 1 个出队的事件
       then p←x,q←work-matrix[i,j]
          if type[key[x]]=1                     ▷若 x 是等待电梯的事件
            then if begin[key[p]]=begin[key[q]]
                   then length[key[p]]←length[key[p]]+5
                        flag←true
          if type[key[x]]=2                     ▷若 x 是等待进入房间的事件
            then t1←begin[key[p]]+length[key[p]]
                 t2←begin[key[next[q]]]+length[key[next[q]]]
                 if t1<t2
                   then length[key[p]]←length[key[p]]+t2-t1
                        flag←true
          if flag=true
            then begin←begin[key[p]]+length[key[p]]
                 p←next[p]
```

```
20              while p≠NIL                    ▷修改行程表中 p 及其后事件的开始时间
21                do begin[key[p]]←begin
22                   if type[key[p]]=1
23                  then t←(begin[key[p]]+length[key[p]]) MOD 5
24                    if t≠0
25                      then length[key[p]]←5-t
26                      else length[key[p]]←0
27                   if type[key[p]]=1 or type[key[p]]=2
28                      then FIX(queues[floor[to[key[p]]],number[to[key[p]]]])
29                   begin←begin[key[p]]+length[key[p]]
30                   p←next[p]
31          work-matrix[i,j]←x
32          DEQUEUE(queues[i,j])
33          x←MINI(queues)
```

算法 3-21　模拟特派员在大楼中活动处理行程表的过程

过程 PROCESS-SCHEDULES 的工作如下。

首先将 *work-matrix* 的所有元素置为空(NIL)。第 2 行调用过程 MINI 计算 *queues* 中各队列的队首最小者并返回给 x。第 3～33 行的 **while** 循环模拟整个大楼中特派员的活动。该循环的每次重复分别对等待电梯和等待进入房间的事件进行处理。其中,第 4 行计算 x 所在等待队列的下标 i 和 j。第 5 行测试 x 是否为该等待队列第 1 次出队的事件。若否,则第 6～30 行对于其进行处理。第 6 行分别用 p、q 指向 x 和 *queues*[i,j]的前一个出队的队首,它保存在 *work-matrix*[i,j]。第 7～10 行处理等待电梯的事件(*type*[*key*[x]]=1)。当 p 的开始时间和本队列中上一次出队的事件 q 的开始时间相等时,p 需要多等 5s。第 11～16 行处理等待进入房间的事件(*type*[*key*[x]]=2)。当 p 完成时间小于 q 的下一个事件(q 从属的特派员完成等待事件 q 之后发生的待在房间中的事件)的完成时间,需要调整 p 的等待时间。此时,p 要等到 *next*[q]完成。无论 p 是哪种类型的事件,只要其等待时间 *length* 发生了调整,就将标志 *flag*(在第 4 行初始化为 *false*)置为 *true*。第 17～30 行对 p 发生了等待时间调整的情形,修改 p 从属的特派员的行程链表中 p 以后各行程事件的开始时间。注意,当处理的是等待电梯事件时,第 27～30 行还需酌情调整其等待时间以保证进入电梯的时间是 5s 的倍数。当处理的是等待事件(无论是等待电梯还是等待进入房间)时,修改开始时间后均应维护所在等待队列,保持优先性质(第 27 行和第 28 行)。无论是否需要调整 x 的等待时间,对 x 的处理结束后,第 31 行将 x 保留到 *work-matrix*[i,j]中,作为队列 *queues*[i,j]中下一个队首比较参照事件。第 32 行将 x 从 *queues*[i,j]弹出,第 33 行重新计算 *queues* 中所有队首的最小者,并返回给 x,作为下一轮的处理事件。

3. 程序实现

1) 数据类型设计及全局量设置

根据对问题中涉及的房间、事件、事件组成的行程表、模拟特派员在大楼内活动时电梯和房间前的等待队列的说明以及事件间的比较规则,有如下类型定义、数据声明及函数定义。

```
1 typedef struct{                                        /* 房间类型 */
2    int floor;                                          /* 楼层 */
3    int number;                                         /* 房间号 */
4 }Room;
5 typedef struct{                                        /* 事件类型 */
6    char code;                                          /* 特派员代码 */
7    int type;                                           /* 事件类型 */
8    Room from;                                          /* 起点房间 */
9    Room to;                                            /* 终点房间 */
10   int begin;                                          /* 开始时间 */
11   int length;                                         /* 持续时间 */
12 }Event;
13 PQueue *queues[11][11];                               /* 电梯与房间等待队列 */
14 LinkedList *schedules[26];                            /* 各特派员的行程表 */
15 FILE *f1, *f2;                                        /* 输入输出文件 */
16 int compare(ListNode **node1,ListNode **node2){       /* 等待队列中事件的比较 */
17    Event *e1=(*node1)->key, *e2=(*node2)->key;        /* 解析出结点所含事件 */
18    if(e1->begin<e2->begin)                            /* e1 的开始时间较小 */
19        return 1;
20    if(e1->begin==e2->begin)                           /* 开始时间相等 */
21        return e2->code-e1->code;
22    return -1;
23 }
```

程序 3-22　表示房间和事件的数据结构

对程序 3-22 的说明如下。

(1) 第 1～4 行和第 5～12 行分别定义了表示房间和事件的数据类型 Room 和 Event。它们的属性数据的意义前文已有说明,此处不再赘述。

(2) 第 13 行声明了等待队列矩阵 queues,这是一个二维数组,共有 11 行 11 列。第 0 列表示各层电梯的等待队列,即 queues[i][0]表示第 i 层楼的等待电梯队列,i=1,2,…,10。由于没有第 0 层,所以 queues[0][j],j=0,1,…,10 弃之不用。这个数组的元素类型是本节开发的基于堆的通用优先队列指针 PQueue *。第 14 行声明了行程链表数组 schedules。每个特派员在大楼中的活动均按其行程表进行,由于一天最多有 26 个特派员进入大楼,所以该数组共有 26 个元素。元素类型是第 2 章开发的通用双向链表指针 LinkedList *。第 15 行声明了两个文件指针 f1 和 f2,表示程序的输入输出文件。

(3) 在等待队列 queues[i][j]中,事件之间需要进行比较。第 16～23 行定义的函数 compare 对两个双重指针 e1 和 e2 指引的事件按题面说明的规则比较大小,用于行程链表中结点比较。等待队列中的事件之所以需要用双重指针指引,是因为它们是所从属的链表中的结点,一旦在处理过程中开始时间被修改,就应该反映在其等待队列中的优先级。用双重指针表示链表中的数据与等待队列中的数据的这种关联关系。

2) **初始化行程表**

要解决本问题首先要根据输入文件中各特派员的行程数据创建他们的行程表,而行程

表是由一系列的事件按事件顺序排列而成的。在程序中需要实现算法 3-18 至算法 3-20 来完成这一任务。为节省篇幅，下面仅列出创建进入大楼事件、创建电梯等待事件及创建行程表 3 个过程的实现代码并加以解析。其他的创建事件的函数代码读者可打开源文件自行解读。

```
int entry(char code,int begin){                        /*进入大楼*/
  Event e={code,0,{0,0},{0,0},begin,30};
  schedules[code-'A']=createList(sizeof(Event),NULL);
  listPushBack(schedules[code-'A'],&e);
  return e.begin+e.length;
}
int waitElevator(char code,int begin,int floor){       /*等待电梯*/
  Event e={code,1,{floor,0},{floor,0},begin,begin%5?5-begin%5:0};
  listPushBack(schedules[code-'A'],&e);
   if(queues[floor][0]==NULL)
       queues[floor][0]=initPQueue(sizeof(ListNode*),26,compare);
   enQueue(queues[floor][0],&(schedules[code-'A']->nil->prev));
   return e.begin+e.length;
}
void initSchedules(){
   char code,room[5],floor[3];
   int hour,minute,second,length,begin;
   Room from,to;
   /*读入特派员代码及来到大楼的时间*/
   fscanf(f1,"%c %d:%d:%d",&code,&hour,&minute,&second);
   while(code!='.'){
     begin=hour*3600+minute*60+second;          /*换算开始时间*/
     begin=entry(code,begin);                   /*创建并在行程表中加入进入大楼事件*/
     from.floor=1,from.number=0;
     fscanf(f1,"%s %d",room,&length);           /*读入该特派员下一项行程数据*/
     to.floor=atoi(strncpy(floor,room,2));      /*解析目标房间楼层*/
     to.number=atoi(strncpy(floor,room+2,2));   /*解析目标房间号码*/
     while(strlen(room)>1){
       begin=toRoom(code,begin,length,from,to); /*处理一项行程数据*/
       from=to;                                 /*设置下一项起始地点*/
       fscanf(f1,"%s %d",room,&length);         /*读取下一项行程数据*/
       to.floor=atoi(strncpy(floor,room,2));
       to.number=atoi(strncpy(floor,room+2,2));
     }
     Exit(code,begin,from);                     /*该特派员离开大楼*/
                                                /*读取下一个特派员的数据*/
       fscanf(f1,"%c %d:%d:%d",&code,&hour,&minute,&second);
   }
}
```

程序 3-23　创建行程表的 C 函数

对程序 3-23 的说明如下。

(1) 第 1～6 行定义的函数 entry 实现算法 3-18 中的 ENTRY 过程。由于把事件表示为结构体对象,C 中结构体变量是可以初始化的,所以初始化事件 e 的代码甚至比伪代码还简洁:第 2 行等价于算法过程中第 1～6 行整整 6 行。由于进入大楼事件是特派员 code 的行程表中的第一个事件,所以第 3 行调用函数 createList[1] 为 schedules[code-'A']创建一个空的双向链表,第 4 行调用函数 listPushBack 将事件 e 追加到链表 schedules[code－'A']的末尾。注意,code 的值为'A'～'Z',故 code－'A'对应 0～26。

(2) 第 7～14 行定义的函数 waitElevator 实现算法 3-18 中的 WAIT-ELEVATOR 过程。由于等待电梯事件之前必发生过其他事件(至少发生过进入大楼时间),所以,在本过程中无须创建链表,而在第 9 行直接将初始化后的事件 e 追加到链表 schedules[code－'A']的末尾。由于 e 是等待事件,故须在第 12 行调用函数 enQueue[2] 将其所在的链表 schedules[code－'A']中对应的结点(尾部结点 schedules[code－'A']->nil->prev)加入到由参数 floor 确定的楼层的电梯等待队列 queues[floor][0]中。不过由于这是第一次在该队列中加入元素,故第 10 行和第 11 行先检测 queues[floor][0]是否为 NULL,若是,需先调用函数 initPQueue 创建一个优先队列。

(3) 第 15～39 行定义的函数 initSchedules 实现算法 3-20 的 INIT-SCHEDULES 过程。由于把输入文件定义成全局量,所以省略了参数 f。函数的代码与算法过程的伪代码结构十分接近。要注意的情况如下。

① 从输入文件读取的开始时间格式为 hh:mm:ss,因此,需要在第 22 行将其转换成以秒为单位的整数 begin。

② 输入文件中表示房间的数据格式是 xxyy 的串,所以第 26 行和第 27 行(同样的第 32 行和第 33 行)通过调用库函数 strncpy[3] 析取串 room 中的子串 xx 和 yy,并调用库函数 atoi 将它们转换成整数赋予表示目标房间的 Room 型变量 to 的 floor 和 number 属性。

③ 输入文件中一个特派员行程数据的结束标志是仅含 0 的一行,所以,嵌套在内部的第 28～34 的 **while** 循环的结束条件是 strlen(room)＝＝1。

3) 处理行程表

对特派员们在大楼中活动的行程表进行处理的 PROCESS-SCHEDULES 过程是解决 Department 问题的核心。下列函数实现该过程。

```
void processSchedules(){                              /* 处理行程表 */
  ListNode * workMatrix[11][11]={NULL};               /* 工作矩阵 */
  int i,j;
  ListNode * x;
  while(x=mini(&i,&j)){
    if(workMatrix[i][j]){                             /* 不是第一次出队 */
      int flag=0;                                     /* 维持时间调整标志,初始化为 false */
      char code=((Event * )(x->key))->code;           /* 确定特派员 */
```

① 此处的函数 listPushBack 和下文中的函数 listPushBack 的定义见第 2.1.3 节的程序 2-3 和程序 2-4。

② 此处的函数 enQueue 及下文中的函数 initPQueue 的定义见 3.4.2 节的程序 3-20 和程序 3-19。

③ 此处的库函数 strncpy 及下文中的库函数 atoi 的原型声明都在头文件 string.h 中。

```
9       ListNode *p=x, *q=workMatrix[i][j];
10      if(((Event *)(x->key))->type==1){             /* 是等待电梯事件 */
11        Event *e1=p->key, *e2=q->key;
12        if(e1->begin==e2->begin){
13          e1->lenth=e2->lenth+5;
14          flag=1;
15        }
16      }
17      if(((Event *)(x->key))->type==2){             /* 是等待进入房间事件 */
18        Event *e1=p->key, *e2=q->next->key;
19        int t1=e1->begin+e1->lenth,                  /* 本队列本次事件完成时间 */
20            t2=e2->begin+e2->lenth;                  /* 上次事件完成时间 */
21        if(t1<t2){
22            e1->lenth+=t2-t1;
23            flag=1;
24        }
25      }
26        if(flag){                                    /* 等待时间做了调整 */
27        Event *e=p->key;
28        int begin=e->begin+e->lenth;                 /* 下次事件的开始时间 */
29        for(p=p->next,e=p->key;p!=schedules[code-'A']->nil;
                p=p->next,e=p->key){
30          e->begin=begin;                            /* 修改开始时间 */
31          if(e->type==1){                            /* 是等待电梯事件 */
32            int t=begin%5;
33            e->lenth=t?5-t:0;
34          }
35          begin=e->begin+e->lenth;
36          if(e->type==1 || e->type==2)               /* 是等待事件 */
37              fix(queues[e->to.floor][e->to.number]); /* 维护所在等待队列 */
38          }
39        }
40      }
41      workMatrix[i][j]=x;
42      deQueue(queues[i][j]);
43    }
44 }
```

程序 3-24　实现行程表处理过程的 C 函数

对程序 3-24 的说明如下。

(1) 由于等待队列队首最小者 x、指向 x 的指针 p、指向 x 所在队列 queues[i][j]上一次出队队首 workeMatrix[i][j]的指针的基类型都是 ListNode，其属性 key 才指向某个事件。为使代码简洁易读，第 10～16 行中引入指向 p、q 的 key 属性的指针 e1、e2，第 17～25 行引入指向 p、q->next 的 key 属性的指针 e1 和 e2。第 29～38 行引入指向 p 的 key 属性指针 e。

(2) 第 5 行调用函数 mini 计算并返回各等待队列中队首的最小者给 x,若所有的等待队列中都无元素了,函数返回 NULL。所以,可以将此赋值表达式作为第 5～43 行的 **while** 循环的循环条件。传递给 mini 函数的参数是变量 i、j 的地址,mini 在计算等待队列队首最小者的同时还跟踪最小者所在队列的下标。由于队列构成的是二维数组,主调函数可通过这两个参数得到这两个数据。这样就免去了算法过程中第 4 行计算 x 所在队列下标 i、j 的操作。mini 函数定义如下。

```
ListNode * mini(int * row,int * col){                /* 计算并返回各等队列队首中的最小者 */
  Event e={'a',-1,{0,0},{0,0},INT_MAX,0};            /* 用来创建链表结点的特殊事件 */
  ListNode n={&e,NULL,NULL}, * x=&n;
  int i,j;
  for( * row=0,i=1;i<11;i++)
    for(j=0;j<11;j++)
      if(queues[i][j]&&!empty(queues[i][j])&&
         compare(top(queues[i][j]),&x)>0){           /* queues[i][j]的队首小于 x */
        x= * (ListNode**)(top(queues[i][j]));
        * row=i;                                     /* 跟踪 x 所在队列的下标 */
        * col=j;
      }
   if( * row)                                        /* 有效的最小值 */
       return x;
   return NULL;                                      /* queues 中所有的队列均为空 */
}
```

程序 3-25 计算各等待队列队首最小者的 C 函数

(3) 程序 3-24 中的第 26～39 行对应算法 3-21 中第 17～30 行,对某等待事件的等待时间做了变更后在其所在的行程表中以后结点修改开始时间的操作。对行程表中所处理的每个结点,第 36 行测得其为等待事件时,由于它的优先级发生了变化,需要调用函数 fix 对其所在等待队列维护堆性质。fix 函数对指定优先队列的保存堆的数组 heap 调用本节程序 3-17 中定义的 buildHeap 函数,重建堆。其定义如下。

```
void fix(PQueue * q){
    buildHeap(q->heap,q->eleSize,q->heapSize,q->compare);
}
```

把函数 fix 的定义添加到文件夹 datastructure 的源文件 pqueue.c 中,并将其原型声明添加到同一文件夹内的头文件 pqueue.h 中。已备今后使用。

4) 输出行程表

对处理好的各个特派员的行程表,用下列函数输出。

```
void printSchedules(){
  int i;
  for(i=0;i<26;i++)
      if(schedules[i]){
          fprintf(f2,"%c\n",'A'+i);
```

```
            listTraverse(schedules[i],printEvent);
            fprintf(f2,"\n");
            clrList(schedules[i],NULL);
            free(schedules[i]);
    }
}
```

程序 3-26　输出行程表的 C 函数

函数 printSchedules 对非空的链表 schedules[i]调用 listTraverse 对其进行遍历操作。注意，传递给 listTraverse 的第 2 个参数是遍历链表时对每个结点的操作函数指针 printEvent。该函数负责输出结点中所含的事件信息。定义如下。

```
void printEvent(Event * e){                              /* 输出事件 */
  int hour,minute,second,t=e->begin;
  if(e->length==0)                                       /* 持续时间为 0s */
      return;
  hour=t/3600;t=t%3600;minute=t/60;second=t%60;
  fprintf(f2,"%02d:%02d:%02d ",hour,minute,second);      /* 输出开始时间 */
  t=e->begin+e->length;
  hour=t/3600;t=t%3600;minute=t/60;second=t%60;
  fprintf(f2,"%02d:%02d:%02d ",hour,minute,second);      /* 输出完成时间 */
   switch(e->type){                                      /* 按事件类型输出相关信息 */
        case 0: fprintf(f2,"Entry\n"); break;
        case 1: fprintf(f2,"Waiting in elevator queue\n"); break;
        case 2:fprintf(f2,"Waiting in front of room %02d%02d\n",
                    e->from.floor,e->from.number);
            break;
        case 3:fprintf(f2,"Transfer from room %02d%02d to room %02d%02d\n",
                    e->from.floor,e->from.number,e->to.floor,e->to.number);
            break;
        case 4:fprintf(f2,"Transfer from elevator to room %02d%02d\n",
                    e->to.floor,e->to.number);
            break;
        case 5:fprintf(f2,"Transfer from  room %02d%02d to elevator\n",
                    e->from.floor,e->from.number);
            break;
        case 6:fprintf(f2,"Stay in room %02d%02d\n",
                    e->from.floor,e->from.number);
            break;
        case 7:fprintf(f2,"Stay in elevator\n");break;
        case 8:fprintf(f2,"Exit\n");
   }
}
```

程序 3-27　输出事件信息的 C 函数

该函数对持续时间 length 大于 0 的事件,按题面要求的格式向输出文件输出事件信息。它通过一个 switch 语句,对不同类型的事件,输出不同的信息。

调用函数 initSchedules、processSchedules 和 printSchedules 解决 Department 问题的 main 函数以及尚未罗列出来的其他功能函数都存储在文件夹 chap03/Department 内的源文件 department.c 中,读者可打开研读

第4章 代数计算

数学是科学的皇后，计算科学当然也不例外。计算科学不但以数学为基础，并且以解决数学计算问题为己任。事实上，无论是科学、技术或是日常生活，人们无时无刻不需要运用数学的知识、方法解决所面临问题中的计算。计算机是计算的利器，使用计算机来帮助人们解决数学计算问题是实至名归。

学过数学的人都知道，代数和几何是数学学科的基础。本章运用第3章讨论的基本算法设计策略解决代数学中的几个基本问题，第5章讨论解决计算几何中的几个经典问题的算法。这些关于数学的算法不仅本身饶有兴味，它们也是本书以后各章所要解决的一些经典问题所需的基础算法。我们还将把一些广泛应用的数学计算算法实现为通用的程序，以备调用。

4.1 矩阵及其计算

4.1.1 矩阵与向量

矩阵是由数构成的矩形列阵。例如，

$$\mathbf{A}=\begin{pmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\end{pmatrix}=\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} \tag{4-1}$$

是一个 2×3 矩阵，$\mathbf{A}=(a_{ij})$，其中 $i=1,2$ 及 $j=1,2,3$，矩阵的第 i 行第 j 列元素是 a_{ij}。用大写字母表示矩阵并用对应的带有下标的小写字母表示该矩阵的元素。元素为实数的所有 $m\times n$ 矩阵表为 $\boldsymbol{R}^{m\times n}$。一般地，元素取自集合 $\boldsymbol{S}$ 的 $m\times n$ 矩阵的全体记为 $\boldsymbol{S}^{m\times n}$。在计算机中，矩阵 $\mathbf{A}$ 的行数和列数表示为属性 $rows[A]$ 和 $columns[A]$。

矩阵 $\mathbf{A}$ 的**转置**是由 $\mathbf{A}$ 的行(列)转换为列(行)而得的矩阵 $\mathbf{A}^T$。对式(4-1)中的矩阵 $\mathbf{A}$，

$$\mathbf{A}^T=\begin{pmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{pmatrix}$$

实现矩阵转置的算法过程描述如下。

```
TRANSFORM(A)
m←rows[A], n←columns[A]
for i←1 to m
    do for j←1 to n
        do A^T[i,j]←A[j,i]
return A^T
```

算法 4-1　计算矩阵的转置过程

向量是一个一维数组。例如,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{4-2}$$

是一个长度为 3 的向量。用小写字母来表示向量,并将长度为 n 的向量 $\boldsymbol{x}$ 中的第 i 个元素表示为 x_i,$i=1,2,\cdots,n$。把等价于一个 $n\times1$ 矩阵的**列向量**作为标准形式,而相应的**行向量**是对列向量取其转置而的:

$$\boldsymbol{x}^T = (2,3,5)$$

单位向量 $\boldsymbol{e}_i$ 是第 i 个元素为 1,其他元素均为 0 的向量。通常,单位向量的长度是很容易由上下文而知的。

零矩阵是所有元素均为 0 的矩阵。这样的矩阵常表示为 **0**,这是因为它与数 0 之间的含混很容易从上下文得以澄清。若 **0** 表示的是矩阵,则其规模(行数与列数)也需要从上下文中推得。

$n\times n$ **方阵**是常见的。方阵的几种特殊情形是很有趣的。

(1) **对角阵**中只要 $i\neq j$ 就有 $a_{ij}=0$。由于非对角线上的元素均为零,可以用对角线元素的列表来表示该矩阵:

$$\text{diag}(a_{11},a_{22},\cdots,a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

(2) $n\times n$ **单位阵** $\boldsymbol{I}_n$ 是对角线上元素均为 1 的对角阵:

$$\boldsymbol{I}_n = \text{diag}(1,1,\cdots,1) = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

当单位阵 $\boldsymbol{I}$ 没有下标表示时,其规模也要从上下文中推得。单位阵的第 i 列是单位向量 $\boldsymbol{e}_i$。

(3) $n\times n$ **上三角阵** $\boldsymbol{U}$ 中,若 $i>j$ 就有 $u_{ij}=0$。对角线以下所有元素均为 0:

$$\boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

上三角阵中若对角线上的元素均为 1,称为**单位上三角阵**。

(4) $n\times n$ **下三角阵** $\boldsymbol{L}$ 中,若 $i<j$ 就有 $l_{ij}=0$。对角线以上的元素均为零:

$$\boldsymbol{L} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}$$

若下三角阵的对角线元素均为 1,称为**单位下三角阵**。

(5) **置换阵** $\boldsymbol{P}$ 中的每一行或每一列仅有一个元素为 1,其余元素均为 0。置换阵的一个

例子是：

$$P=\begin{pmatrix}0&1&0&0&0\\0&0&0&1&0\\1&0&0&0&0\\0&0&0&0&1\\0&0&1&0&0\end{pmatrix}$$

之所以把这样的矩阵称为置换阵，是因为将其与一向量 $\boldsymbol{x}$ 相乘，具有将 $\boldsymbol{x}$ 的元素做置换（重排）的效果。

（6）**对称阵** $\mathbf{A}$ 满足条件 $\mathbf{A}=\mathbf{A}^T$。例如，

$$\begin{pmatrix}1&2&3\\2&6&4\\3&4&5\end{pmatrix}$$

就是一个对称阵。

4.1.2　矩阵的运算

矩阵的元素取自于一个诸如实数系、复数系，或以素数为模的整数数系。数系中定义了数的加法和乘法，可以把包括加法和乘法的定义扩展到矩阵上。

定义矩阵的**加法**如下。若 $\mathbf{A}=(a_{ij})$ 和 $\mathbf{B}=(b_{ij})$ 是 $m\times n$ 矩阵，则它们的**矩阵和** $\mathbf{C}=(c_{ij})=\mathbf{A}+\mathbf{B}$ 定义为 $m\times n$ 矩阵：

$$c_{ij}=a_{ij}+b_{ij}$$

$i=1,2,\cdots,m$ 及 $j=1,2,\cdots,n$。也就是说，矩阵的加法是按元素进行的。写成伪代码过程如下。

```
MATRIX-ADD(A,B)
1 m←row[A],n←colum[A]
2 if row[B]≠m or colum[B]≠n
3     then error "A,B 加法不相容"
4 for i←1 to m
5     do for j←1 to n
6          do c_ij←a_ij+b_ij
7 return C        ▷C=(c_ij)_m×n
```

算法 4-2　矩阵的加法过程

该算法的运行时间为 $\Theta(mn)$。

零矩阵是矩阵加法的单位元：

$$\mathbf{A}+\mathbf{0}=\mathbf{A}=\mathbf{0}+\mathbf{A}$$

若 λ 是一个数且 $\mathbf{A}=(a_{ij})$ 是一个矩阵，则 $\lambda\mathbf{A}=(\lambda a_{ij})$，也就是将 $\mathbf{A}$ 的每个元素与 λ 的积构成的矩阵称为 $\mathbf{A}$ 的**标量积**。作为一个特殊情形，定义矩阵 $\mathbf{A}=(a_{ij})$ 的**负阵**为 $-1\cdot\mathbf{A}=-\mathbf{A}$，即 $-\mathbf{A}$ 的第 ij 个元素是 $-a_{ij}$。于是，

$$\mathbf{A}+(-\mathbf{A})=\mathbf{0}=(-\mathbf{A})+\mathbf{A}$$

根据此定义，人们把**矩阵差**定义为负阵的和：$\boldsymbol{A}-\boldsymbol{B}=\boldsymbol{A}+(-\boldsymbol{B})$。

矩阵积的定义如下。两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是**乘法相容**的，若 $\boldsymbol{A}$ 的列数等于 $\boldsymbol{B}$ 的行数（一般地，若表达式中含有矩阵积 $\boldsymbol{AB}$，总是假定此意味着 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是乘法相容的）。若 $\boldsymbol{A}=(a_{ij})$ 是一个 $m\times n$ 矩阵，$\boldsymbol{B}=(b_{jk})$ 是一个 $n\times p$ 矩阵，则它们的矩阵积 $\boldsymbol{C}=\boldsymbol{AB}$ 是 $m\times p$ 矩阵，$\boldsymbol{C}=(c_{ik})$，其中，

$$c_{ik}=\sum_{j=1}^{n}a_{ij}b_{jk} \tag{4-3}$$

$i=1,2,\cdots,m$ 及 $k=1,2,\cdots,p$。下列的过程 MATRIX-MULTIPLY 实现了按式（4-3）的矩阵直接乘法。

```
MATRIX-MULTIPLY(A,B)
if columns[A] ≠ rows[B]
    then error "矩阵乘法不相容"
    else for i←1 to rows[A]
        do for j←1 to columns[B]
            do C[i,j]←0
                for k←1 to columns[A]
                    do C[i,j]←C[i,j]+A[i,k]·B[k,j]
return C
```

算法 4-3　矩阵的乘法过程

为将 $n\times n$ 矩阵相乘，MATRIX-MULTIPLY 执行 n^3 次乘法和 $n^2(n-1)$ 次加法，所以其运行时间是 $\Theta(n^3)$。

矩阵有许多（但不是所有）与数相同的经典代数性质。单位阵是矩阵乘法的单位元：

对任意的 $m\times n$ 矩阵 $\boldsymbol{A}$：

$$\boldsymbol{I}_m\boldsymbol{A}=\boldsymbol{A}\boldsymbol{I}_n=\boldsymbol{A}$$

乘以零矩阵得零矩阵：

$$\boldsymbol{A}\boldsymbol{0}=\boldsymbol{0}$$

矩阵乘法是结合的，对相容的矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$ 和 $\boldsymbol{C}$：

$$\boldsymbol{A}(\boldsymbol{BC})=(\boldsymbol{AB})\boldsymbol{C} \tag{4-4}$$

矩阵乘法对加法是分配的：

$$\boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C})=\boldsymbol{AB}+\boldsymbol{AC}$$
$$(\boldsymbol{B}+\boldsymbol{C})\boldsymbol{A}=\boldsymbol{BA}+\boldsymbol{CA} \tag{4-5}$$

对 $n>1$，$n\times n$ 矩阵的乘法不是交换的。例如，若 $\boldsymbol{A}=\begin{pmatrix}0&1\\0&0\end{pmatrix}$ 及 $\boldsymbol{B}=\begin{pmatrix}0&0\\1&0\end{pmatrix}$，则 $\boldsymbol{AB}=\begin{pmatrix}1&0\\0&0\end{pmatrix}$ 及 $\boldsymbol{BA}=\begin{pmatrix}0&0\\0&1\end{pmatrix}$。

矩阵-向量积或向量-向量积定义为向量是 $n\times 1$ 矩阵（或当向量是行向量时为 $1\times n$ 矩阵）。于是，若 $\boldsymbol{A}$ 是 $m\times n$ 矩阵而 x 是长度为 n 的向量，则 $\boldsymbol{Ax}$ 是长度为 m 的向量。若 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 是长度为 n 的向量，则

$$\boldsymbol{x}^T\boldsymbol{y}=\sum_{i=1}^{n}x_iy_i$$

是一个数(即为 1×1 矩阵),称为 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的**内积**。$\boldsymbol{x}\boldsymbol{y}^T$ 是一个矩阵 $\boldsymbol{Z}$,称为 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的外积,其中 $z_{ij}=x_i y_j$。长度为 n 的向量 $\boldsymbol{x}$(**欧几里德**)**范数** $\|\boldsymbol{x}\|$ 定义为

$$\|\boldsymbol{x}\| = (x_1^2+x_2^2+\cdots+x_n^2)^{1/2} = (\boldsymbol{x}^T\boldsymbol{x})^{1/2}。$$

于是,$\boldsymbol{x}$ 的范数是其在 n 维欧几里德空间中的长度。

4.1.3　矩阵的性质

定义一个 $n\times n$ 矩阵 $\boldsymbol{A}$ 的**逆**是另一个 $n\times n$ 矩阵,记为 $\boldsymbol{A}^{-1}$(若存在),使得 $\boldsymbol{A}\boldsymbol{A}^{-1}=\boldsymbol{I}_n=\boldsymbol{A}^{-1}\boldsymbol{A}$。例如,

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{-1} = \begin{pmatrix}0 & 1\\ 1 & -1\end{pmatrix}$$

很多非零 $n\times n$ 矩阵没有逆。一个没有逆的矩阵称为**不可逆**,或**奇异**的。非零奇异矩阵的一个例子是

$$\begin{pmatrix}1 & 0\\ 1 & 0\end{pmatrix}$$

若一个矩阵有逆,称其为**可逆**的,或**非奇异**的。如果一个矩阵有逆,则其逆是唯一的。若 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是非奇异的 $n\times n$ 矩阵,则

$$(\boldsymbol{B}\boldsymbol{A})^{-1} = \boldsymbol{A}^{-1}\boldsymbol{B}^{-1} \tag{4-6}$$

求逆运算和转置运算是可交换的:

$$(\boldsymbol{A}^{-1})^T = (\boldsymbol{A}^T)^{-1}$$

向量组 $\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_n$ 是**线性相关**的,若有一组不全为零的系数 $c_1,c_2,\cdots,c_n$,使得 $c_1\boldsymbol{x}_1+c_2\boldsymbol{x}_2+\cdots+c_n\boldsymbol{x}_n=0$。例如,行向量 $\boldsymbol{x}_1=(1,2,3)$,$\boldsymbol{x}_2=(2,6,4)$,和 $\boldsymbol{x}_3=(4,11,9)$是线性相关的,这是因为 $2x_1+3x_2-2x_3=0$。若向量组不是线性相关的,则它们是**线性无关**的。例如,组成单位矩阵的各列向量就是线性无关的。

非零 $m\times n$ 矩阵 $\boldsymbol{A}$ 的**列秩**是其列向量的最大线性无关组中的向量个数。类似地,$\boldsymbol{A}$ 的**行秩**是其行向量的最大线性无关组中的向量个数。任一矩阵 $\boldsymbol{A}$ 的一个基本性质是其行秩总是等于其列秩,因此可以直接引用 $\boldsymbol{A}$ 的**秩**。一个 $m\times n$ 矩阵的秩是一个介于 0 与 $\min(m,n)$ 之间的整数(零矩阵的秩是 0,单位阵的秩是 n)。非零 $m\times n$ 矩阵 $\boldsymbol{A}$ 的秩的另一个等价并且是更有用的定义,它是使

$$\boldsymbol{A} = \boldsymbol{B}\boldsymbol{C}$$

的 $m\times r$ 矩阵 $\boldsymbol{B}$ 和 $r\times n$ 矩阵 $\boldsymbol{C}$ 的最小整数 r。

一方阵具有**满秩**,若其秩为 n。一 $m\times n$ 矩阵具有**满列秩**,若其秩为 n。秩的一个基本性质由下列定理给出。

定理 4-1　一方阵具有满秩当且仅当它是非奇异的。

矩阵 $\boldsymbol{A}$ 的**零向量**是一个非零向量 $\boldsymbol{x}$ 使得 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{0}$。下列的定理及其推论揭示了列秩概念、奇异性与空向量的关系。

定理 4-2　矩阵 $\boldsymbol{A}$ 具有满列秩当且仅当它没有零向量。

推论 4-1　方阵 $\boldsymbol{A}$ 是奇异的当且仅当它有零向量。

$n\times n(n>1)$矩阵 $\boldsymbol{A}$ 的第 ij **子阵** $\boldsymbol{A}_{[ij]}$ 是一个$(n-1)\times(n-1)$矩阵,它是删掉 $\boldsymbol{A}$ 的第 i

行、第 j 列而得到的。$n\times n$ 矩阵 $\boldsymbol{A}$ 的**行列式**利用它的子阵递归定义如下:

$$\det(\boldsymbol{A}) = \begin{cases} a_{11} & n = 1 \\ \sum_{j=1}^{n}(-1)^{1+j}a_{1j}\det(\boldsymbol{A}_{[1,j]}) & n > 1 \end{cases} \tag{4-7}$$

项 $(-1)^{i+j}\det(\boldsymbol{A}_{[ij]})$ 称为元素 a_{ij} 的**余因子**。

下列两条定理表达了行列式的基本性质,它们的证明在此忽略。

定理 4-3(行列式性质) 方阵 $\boldsymbol{A}$ 的行列式有如下性质。

(1) $\boldsymbol{A}$ 的任一行或任一列为零,则 $\det(\boldsymbol{A})=0$。

(2) 若 $\boldsymbol{A}$ 的一行(或一列)的所有元素乘以 λ,则其行列式乘以 λ。

(3) 若将 $\boldsymbol{A}$ 的一行(或一列)的元素加到另一行(或另一列)的对应元素上,行列式不变。

(4) $\boldsymbol{A}$ 的行列式等于 $\boldsymbol{A}^T$ 的行列式。

(5) 若交换 $\boldsymbol{A}$ 的两行(或两列),行列式乘以 -1。

此外,对任意的方阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$,有 $\det(\boldsymbol{AB})=\det(\boldsymbol{A})\det(\boldsymbol{B})$。

定理 4-4 $n\times n$ 矩阵 $\boldsymbol{A}$ 是奇异的当且仅当 $\det(\boldsymbol{A})=0$。

4.1.4 矩阵的程序实现

1. 数据类型的定义

首先,需要定义矩阵的数据类型以及对矩阵进行创建、空间清理等维护的函数。

```
typedef struct{                                    /* 矩阵类型定义 */
    double *tab;                                   /* 二维数表 */
    int row,col;                                   /* 行数、列数 */
}Matrix;
Matrix newMatrix(int m,int n){                     /* 生成 0 矩阵 */
    Matrix a;
    assert(m>=0&&n>=0);
    a.row=m; a.col=n;
    assert(a.tab=(double *)calloc(m*n,sizeof(double)));
    return a;
}
Matrix newMatrixByArry(double * a,int m,int n){    /* 用数组数据生成矩阵 */
    Matrix b=newMatrix(m,n);
    memcpy(b.tab,a,n*m*sizeof(double));
    return b;
}
void clrMatrix(Matrix a){                          /* 清理矩阵空间 */
    free(a.tab);
}
```

程序 4-1 矩阵类型定义及常用维护函数

程序 4-1 的说明如下。

（1）第 1～4 行将矩阵定义为具有 3 个属性的结构体类型 Matrix。**int** 型属性 row 和 col 分别表示矩阵的行数和列数。**double** 型指针属性 tab 表示矩阵的二维数表。之所以用一维指针表示二维数表，出于两个方面的考虑：首先，为一维指针分配存储空间的代码比较简洁。其次，计算一维数组元素下标比计算二维数组元素下标的效率高。我们约定：按行优先原则将二维数表存储于一维数组中。

（2）第 5～11 行定义的函数 newMatrix 用两个 **int** 型参数 m 和 n 创建一个 m×n 的矩阵，作为函数值返回。由于第 9 行调用动态内存分配函数 calloc，创建的矩阵数表元素被初始化为 0，所以创建的是一个 m×n 的 **0** 矩阵。

（3）第 12～16 行定义的函数 newMatrixByArry 用参数 a（double 型指针指引的数组）中的数据，创建一个由参数 m、n 确定行数和列数的矩阵，作为返回值返回。使用该函数需要注意的是，a 中数据应按行优先原则存储矩阵的数据。例如，用数组 a[9]＝{1，2，3，4，5，6，7，8，9}，调用 newMatrixByarry（a，3，3）创建矩阵 $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$。

（4）第 17～19 行定义的函数 clrMatrix 对 Matrix 型参数 a 的 table 属性指引的动态空间执行 free 操作。对行将丢弃不用的 Matrix 型变量，应调用该函数释放占用的存储空间，以防内存泄漏。

2. 矩阵运算实现

定义了 Matrix 类型后，用如下的函数实现算法 4-1 至算法 4-3。

```
Matrix transform(Matrix a){                    /* 计算矩阵的转置 */
    int i,j,m=a.col,n= a.row;
    Matrix at=newMatrix(m,n);
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            at.tab[i*n+j]=a.tab[j*m+i];
    return at;
}
Matrix matixAdd(Matrix a,Matrix b){            /* 矩阵相加 */
    Matrix c;
    int i,j,m=a.row,n=a.col;
    assert(b.row==m&&b.col==n);
    c=newMatrix(m,n);
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            c.tab[i*n+j]=a.tab[i*n+j]+b.tab[i*n+j];
    return c;
}
Matrix matrixMultiply(Matrix a,Matrix b){      /* 矩阵相乘 */
    Matrix c;
    int i,j,k;
```

```
22      assert(a.col==b.row);
23      c=newMatrix(a.row,b.col);
24      for(i=0;i<a.row;i++)
25          for(j=0;j<b.col;j++)
26              for(k=0;k<a.col;k++)
27                  c.tab[i*c.col+j]+=a.tab[i*a.col+k]*b.tab[k*b.col+j];
28      return c;
29 }
```

程序 4-2　实现矩阵加法和乘法运算的函数

程序 4-2 的说明如下。

(1) 第 1～8 行定义的函数 transform 实现算法 4-1,计算 Matrix 型参数 a 的转置矩阵作为函数值返回。函数代码与算法伪代码的结构非常近似,仅指出第 6 行中对 at 的第 i 行第 j 列元素的访问形式是 at.tab[i*n+j],而对 a 的第 j 行第 i 列元素的访问形式为 a.tab[j*m+i],这是用一维数组按行优先原则存储二维数组时,由下标的换算关系决定的。

(2) 第 9～18 行定义的函数 matixAdd 实现算法 4-2,执行 Matrix 型参数 a 和 b 的加法将和作为函数值返回。第 12 行调用库函数 assert,保证矩阵 a 和 b 的行数和列数相同。第 14～16行的两重 **for** 循环实现算法 4-2 中的第 4～6 行的两重 **for** 循环。请注意用一维数组按行优先原则存储二维数组时对各矩阵元素的访问形式。

(3) 第 19～29 行定义的函数 matrixMultiply 实现算法 4-3,计算 Matrix 型参数 a 和 b 的乘法,并将积作为函数值返回。第 22 行调用库函数 assert 保证矩阵 a、b 是乘法相容的。函数的代码与算法伪代码的结构十分近似,读者可比较研读。

程序 4-1 和程序 4-2 中定义的数据类型和各函数的原型声明保存在文件夹 algebra 中的头文件 matrix.h 中,函数定义则保存在同一文件夹的源文件 matrix.c 中,以备在应用中调用。

应用中可能还要进行矩阵间的减法运算,即两个行数、列数相同的矩阵对应元素进行减法运算得到的结果称为两者之差。很容易比照算法 4-2 写一个矩阵相减的算法,我们已将其实现为函数 matrixDiff,保存于 matrix.c 中(原型声明于 matrix.h 中),读者可打开文件研读。

4.2　矩阵的 LUP 分解

在各种应用中经常要解一组联立的线性方程。本章仅讨论具有 n 个未知数 $x_1,x_2,\cdots,x_n$ 的线性方程组:

$$\begin{cases} a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n=b_1 \\ a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n=b_2 \\ \qquad\qquad\vdots \\ a_{n1}x_1+a_{n2}x_2+\cdots+a_{nn}x_n=b_n \end{cases} \tag{4-8}$$

同时满足方程组的一组 $x_1,x_2,\cdots,x_n$ 值称为该方程组的一个**解**。

线性方程组可以表示成一个矩阵方程,其中的每个矩阵或向量都属于某个数域,通常是实数域 **R**。可以方便地将式(4-8)表示为矩阵-向量方程:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

或等价地,设 $\boldsymbol{A}=(a_{ij})$,$\boldsymbol{x}=(x_i)$,$\boldsymbol{b}=(b_i)$,写成:

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{4-9}$$

其中,$\boldsymbol{A}$ 称为式(4-8)的**系数矩阵**。若 $\boldsymbol{A}$ 是非奇异的,它具有逆 $\boldsymbol{A}^{-1}$,则

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{4-10}$$

是解向量。可以证明此 $\boldsymbol{x}$ 是式(4-9)唯一的解。若有两个解 $\boldsymbol{x}$ 和 $\boldsymbol{x}'$,则 $\boldsymbol{Ax}=\boldsymbol{Ax}'=\boldsymbol{b}$ 且

$$\boldsymbol{x} = (\boldsymbol{A}^{-1}\boldsymbol{A})\boldsymbol{x} = \boldsymbol{A}^{-1}(\boldsymbol{Ax}) = \boldsymbol{A}^{-1}(\boldsymbol{Ax}') = (\boldsymbol{A}^{-1}\boldsymbol{A})\boldsymbol{x}' = \boldsymbol{x}'$$

我们主要讨论 A 是非奇异或(按定理 4-1)$\boldsymbol{A}$ 的秩等于未知数个数 n 的情形。代数中通常用 **Gaussian 消去法**解此线性方程组。首先从各方程减去第 1 个方程的倍数,使得这些方程的第一个变量被移除。然后从第 3 个方程起,各方程减去第 2 个方程,使得在这些方程中消去第 1 个变量和第 2 个变量。这些操作对应于对系数矩阵 $\boldsymbol{A}$ 做对应的操作:将某一行的若干倍加到另一行。继续这一进程,直至得到一个上三角形式矩阵 $\boldsymbol{U}$。而为消去各变量的倍数行构成的矩阵 $\boldsymbol{L}$ 是下三角形的。将此方法转换成对系数矩阵的变换,就是在本节中讨论的对矩阵的 LUP 分解。

4.2.1　LUP 分解法概述

对 $n\times n$ 矩阵 $\boldsymbol{A}$,LUP 分解法是求 3 个 $n\times n$ 矩阵 $\boldsymbol{L}$、$\boldsymbol{U}$ 和 $\boldsymbol{P}$ 使得:

$$\boldsymbol{PA} = \boldsymbol{LU} \tag{4-11}$$

其中:

(1) $\boldsymbol{L}$ 为一单位下三角矩阵。

(2) $\boldsymbol{U}$ 为一上三角矩阵。

(3) $\boldsymbol{P}$ 为一置换矩阵。

称满足式(4-11)的矩阵 $\boldsymbol{L}$、$\boldsymbol{U}$ 和 $\boldsymbol{P}$ 为矩阵 $\boldsymbol{A}$ 的一个 **LUP 分解**。我们将证明任一非奇异矩阵 $\boldsymbol{A}$ 都具有这样的分解。

对矩阵 $\boldsymbol{A}$ 进行 LUP 分解的好处在于线性方程组在矩阵 $\boldsymbol{L}$ 和 $\boldsymbol{U}$ 为三角阵时,解起来很方便。对 $\boldsymbol{A}$ 做 LUP 分解后,可以通过如下只解三角形线性方程组来解方程 $\boldsymbol{Ax}=\boldsymbol{b}$。在 $\boldsymbol{Ax}=\boldsymbol{b}$ 的两边乘以 $\boldsymbol{P}$ 得到方程 $\boldsymbol{PAx}=\boldsymbol{Pb}$,这相当于置换方程组(4-8)。利用分解式(4-11),有

$$\boldsymbol{LUx} = \boldsymbol{Pb}$$

现在可以通过解两个三角形线性方程组来解此方程。设 $\boldsymbol{y}=\boldsymbol{Ux}$,其中 $\boldsymbol{x}$ 是要求的解向量。首先,用称为“前代法”的方法解下三角形方程组:

$$\boldsymbol{Ly} = \boldsymbol{Pb} \tag{4-12}$$

得到未知数向量 $\boldsymbol{y}$,然后用称为“回代法”的方法解上三角线性方程:

$$\boldsymbol{Ux} = \boldsymbol{y} \tag{4-13}$$

得到原方程组 $\boldsymbol{Ax}=\boldsymbol{b}$ 的解向量 $\boldsymbol{x}$。这是因为置换矩阵 $\boldsymbol{P}$ 是可逆的：

$$\boldsymbol{Ax} = \boldsymbol{P}^{-1}\boldsymbol{LUx} = \boldsymbol{P}^{-1}\boldsymbol{Ly} = \boldsymbol{P}^{-1}\boldsymbol{Pb} = \boldsymbol{b}$$

4.2.2 LU 分解

下面先讨论简单的情形：矩阵 $\boldsymbol{A}$ 的主对角线上的元素在分解过程中永不为 0，这样就不需要交换各行而省略了置换矩阵 $\boldsymbol{P}$。这样的操作称为 **LU 分解**。实现此策略的算法是递归的。要对 $n\times n$ 非奇异矩阵 $\boldsymbol{A}$ 构造其 LU 分解。若 $n=1$，则计算完成，这是因为可以选取 $\boldsymbol{L}=\boldsymbol{I}_1$ 和 $\boldsymbol{U}=\boldsymbol{A}$。对 $n>1$，将 $\boldsymbol{A}$ 分割成 4 个部分：

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{v} & \boldsymbol{A}' \end{pmatrix}$$

其中 $\boldsymbol{v}$ 是一个长度为$(n-1)$的列向量，$\boldsymbol{w}^T$ 是一个长度为$(n-1)$的行向量，$\boldsymbol{A}'$是一个$(n-1)\times(n-1)$矩阵。然后，直接验证乘积可知，$\boldsymbol{A}$ 可以分解因子：

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{v} & \boldsymbol{A}' \end{pmatrix} = \begin{pmatrix} 1 & \boldsymbol{0} \\ \boldsymbol{v}/a_{11} & \boldsymbol{I}_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{0} & \boldsymbol{A}' - \boldsymbol{vw}^T/a_{11} \end{pmatrix} \tag{4-14}$$

因子分解的前后矩阵中的 $\boldsymbol{0}$ 分别是长度为 $n-1$ 的列向量和行向量。项 $\boldsymbol{vw}^T/a_{11}$ 是将 $\boldsymbol{v}$ 及 $\boldsymbol{w}$ 的外积除以 a_{11} 而得的$(n-1)\times(n-1)$矩阵，它与被其所减的矩阵 $\boldsymbol{A}'$的规模是一致的。所得结果为$(n-1)\times(n-1)$矩阵：

$$\boldsymbol{A}' - \boldsymbol{vw}^T/a_{11} \tag{4-15}$$

称为 $\boldsymbol{A}$ 的关于 a_{11} 的 **Schur 余子式**。

人们断言，若 $\boldsymbol{A}$ 是非奇异的，则其 Schur 余子式也是非奇异的。为什么？假定$(n-1)\times(n-1)$的 Schur 余子式是奇异的，按定理 4-1，其行秩必小于 $n-1$。由于矩阵

$$\begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{0} & \boldsymbol{A}' - \boldsymbol{vw}^T/a_{11} \end{pmatrix}$$

的底部的第一列元素全为 0，此矩阵底部的 $n-1$ 行的行秩必小于 $n-1$。所以此矩阵本身的行秩必小于 n，于是 $\boldsymbol{A}$ 的秩必小于 n。根据定理 4-1，与推出 $\boldsymbol{A}$ 是奇异的矛盾。

由于 Schur 余子式是非奇异的，可以递归地求它的 LU 分解。设

$$\boldsymbol{A}' - \boldsymbol{vw}^T/a_{11} = \boldsymbol{L}'\boldsymbol{U}'$$

其中 $\boldsymbol{L}'$是单位下三角阵，$\boldsymbol{U}'$是上三角阵。于是，利用矩阵代数，有

$$\begin{aligned} \boldsymbol{A} &= \begin{pmatrix} 1 & \boldsymbol{0} \\ \boldsymbol{v}/a_{11} & \boldsymbol{I}_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{0} & \boldsymbol{A}' - \boldsymbol{vw}^T/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & \boldsymbol{0} \\ \boldsymbol{v}/a_{11} & \boldsymbol{I}_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{0} & \boldsymbol{U}' \end{pmatrix} \\ &= \boldsymbol{LU} \end{aligned}$$

这就给出了 LU 分解(注意由于 $\boldsymbol{L}'$是单位下三角阵，所以 $\boldsymbol{L}$ 也是；由于 $\boldsymbol{U}'$是上三角阵，

所以 $\boldsymbol{U}$ 也是)。

图 4-1 示例了 LU 分解。它展示了一种标准的优化过程,其中的元素“原地”存储在矩阵 $\boldsymbol{A}$ 中,即可以在 a_{ij} 和 $l_{ij}(i>j)$ 或 $u_{ij}(i\leqslant j)$ 之间建立起一个对应,并更新 $\boldsymbol{A}$ 使其在过程终结时同时存储。

当然,若 $a_{11}=0$,这种方法就不适用了。这是因为它将导致被 0 除的错误。而当 Schur 余子式 $\boldsymbol{A}'-\boldsymbol{v}\boldsymbol{w}^T/a_{11}$ 的左上角元素为 0 时,此法也不能正常工作,因为在下一步递归时,要用它来做除法。LU 分解中用来做除法的元素称为**基准元素**,它们位于矩阵 $\boldsymbol{U}$ 的对角线上。之所以要在 LUP 分解中设计置换矩阵 $\boldsymbol{P}$,就是要利用置换避免被 0 除的问题。

图 4-1(a)为矩阵 $\boldsymbol{A}$。图 4-1(b)黑圈中的 $a_{11}=2$ 是基准元素,阴影列是 $\boldsymbol{v}/a_{11}$,阴影行是 $\boldsymbol{w}^T$。$\boldsymbol{U}$ 的元素计算于横线之上,$\boldsymbol{L}$ 的元素位于竖线左边。Schur 余子式矩阵 $\boldsymbol{A}'-\boldsymbol{v}\boldsymbol{w}^T/a_{11}$ 位于右下方。图 4-1(c)为对图 4-1(b)中的 Schur 余子式矩阵进行操作。黑圈中的 $a_{22}=4$ 是基准元素,阴影列和阴影行分别是 $\boldsymbol{v}/a_{22}$ 和 $\boldsymbol{w}^T$(Schur 余子式中的部分)。两条直线分割出了目前算出的 $\boldsymbol{U}$ 的元素(上)、$\boldsymbol{L}$ 的元素(左)和 Schur 余子式(右下)。图 4-1(d)为完成分解(当递归结束时,新的 Schur 余子式的元素 3 成为 $\boldsymbol{U}$ 的组成部分)。图 4-1(e)为因子分解 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{U}$。

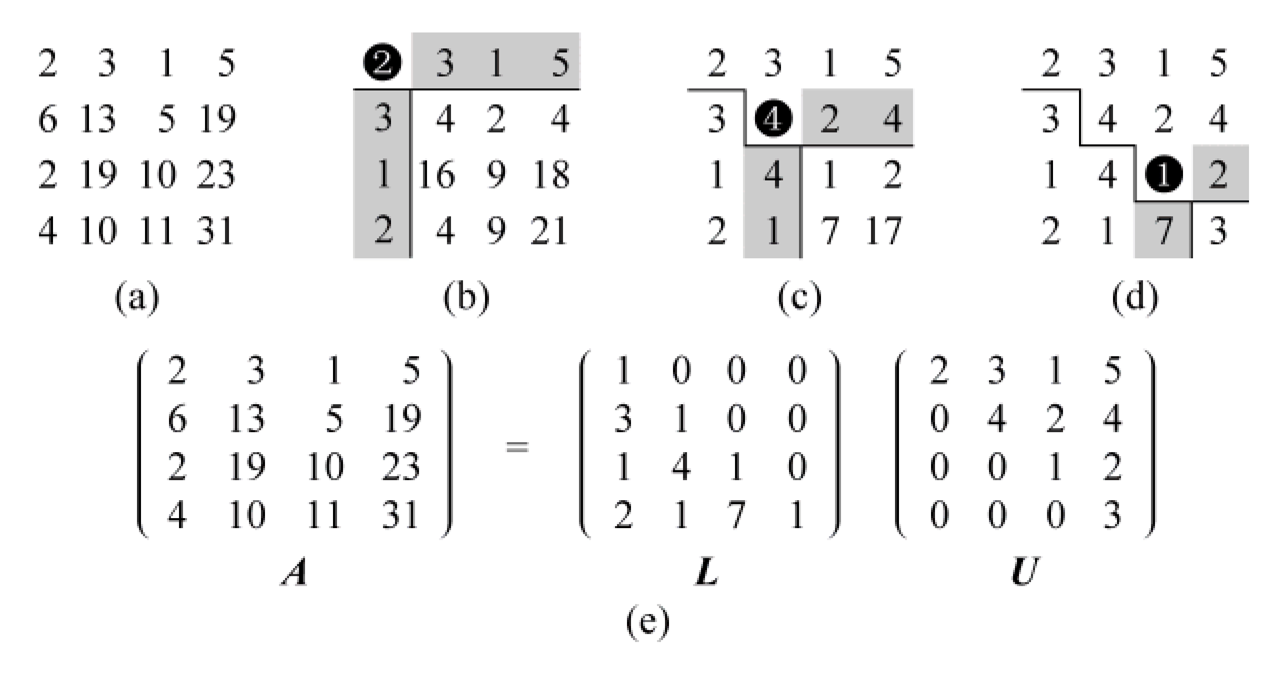


图 4-1　LU 分解

4.2.3　计算 LUP 分解

LUP 分解背后的数学方法类似于 LU 分解。给定 $n\times n$ 矩阵 $\boldsymbol{A}$,希望找到置换矩阵 $\boldsymbol{P}$,单位下三角阵 $\boldsymbol{L}$ 和上三角阵 $\boldsymbol{U}$ 使得 $\boldsymbol{PA}=\boldsymbol{LU}$。在如 LU 分解的那样分割 $\boldsymbol{A}$ 之前,将第一列绝对值最大的非零元素,例如为 a_{k1},移到矩阵的(1,1)位置上(若第一列元素均为 0,按定理 4-4,$\boldsymbol{A}$ 的行列式等于 0,故为奇异的)。为保持方程组,交换第 1 个方程与第 k 个方程,这等价于 $\boldsymbol{A}$ 左乘一个置换矩阵 $\boldsymbol{Q}$。于是,可将 $\boldsymbol{QA}$ 写为

$$\boldsymbol{QA}=\begin{pmatrix} a_{k1} & \boldsymbol{w}^T \\ \boldsymbol{v} & \boldsymbol{A}' \end{pmatrix}$$

其中 $\boldsymbol{v}=(a_{21},a_{31},\cdots,a_{n1})^T$,$a_{11}$ 替换了 a_{k1};$\boldsymbol{w}^T=(a_{k2},a_{k3},\cdots,a_{kn})$;以及 $\boldsymbol{A}'$ 是一个 $(n-1)\times(n-1)$ 矩阵。由于 $a_{k1}\neq 0$,做与在 LU 分解中相同的代数运算,且保证不会有被 0 除的问题:

$$QA=\begin{pmatrix}a_{k1} & w^T\\ v & A'\end{pmatrix}=\begin{pmatrix}1 & \mathbf{0}\\ v/a_{k1} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & A'-vw^T/a_{k1}\end{pmatrix}$$

如在 LU 分解时看到的,若 A 是非奇异的,则 Schur 余子式 $A'-vw^T/a_{k1}$ 也是非奇异的。所以,能递归地对它进行 LUP 分解,得到单位下三角矩阵 L',上三角矩阵和置换矩阵 U',使得:

$$P'(A'-vw^T/a_{k1})=L'U'$$

定义

$$P=\begin{pmatrix}1 & \mathbf{0}\\ \mathbf{0} & P'\end{pmatrix}Q$$

这是一个置换矩阵,这是因为它是两个置换矩阵之积。现在有

$$\begin{aligned}PA&=\begin{pmatrix}1 & \mathbf{0}\\ \mathbf{0} & P'\end{pmatrix}QA=\begin{pmatrix}1 & \mathbf{0}\\ \mathbf{0} & P'\end{pmatrix}\begin{pmatrix}1 & \mathbf{0}\\ v/a_{k1} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & A'-vw^T/a_{k1}\end{pmatrix}\\
&=\begin{pmatrix}1 & \mathbf{0}\\ P'v/a_{k1} & P'\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & A'-vw^T/a_{k1}\end{pmatrix}\\
&=\begin{pmatrix}1 & \mathbf{0}\\ P'v/a_{k1} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & P'(A'-vw^T/a_{k1})\end{pmatrix}\\
&=\begin{pmatrix}1 & \mathbf{0}\\ P'v/a_{k1} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & L'U'\end{pmatrix}\\
&=\begin{pmatrix}1 & \mathbf{0}\\ P'v/a_{k1} & L'\end{pmatrix}\begin{pmatrix}a_{k1} & w^T\\ \mathbf{0} & U'\end{pmatrix}\\
&=LU\end{aligned}$$

得出 LUP 分解。由于 L'是单位下三角矩阵,所以 L 也是;由于 U'是上三角矩阵,所以 U 也是。注意在此推导中,列向量 v/a_{k1} 和 Schur 余子式 $A'-vw^T/a_{k1}$ 都必须与 P'相乘。

很容易将此想法实现为下列的递归过程。动态地用数组 π 维护置换矩阵 P,其中 $\pi[i]=j$ 意味着 P 的第 i 行在第 j 列是 1。初始时 $\pi[1..n]=<1,2,\cdots,n>$,表示单位矩阵。还将代码实现为将 L 及 U"原地"计算在 A 内。于是,当过程终止时(递归层次 k 为 n 时):

$$a_{ij}=\begin{cases}l_{ij} & i>j\\ u_{ij} & i\leqslant j\end{cases}$$

```
LUP-DECOMPOSITION(A,k)
n←rows[A]
if k≥n
    then return
p←0
for i←k to n                          ▷搜寻第 k 列下方绝对值最大的元素
    do if |a_ik|>p
        then p←|a_ik|
             k'←i
    if p=0                            ▷若第 k 列下方元素均为 0
       then error "singular matrix"
```

```
11      exchange π[k]↔π[k']              ▷a_{k'k}是第 k 列下方元素中绝对值最大者
12      for i←1 to n                     ▷交换第 k 行与第 k'行
13          do exchange a_{ki}↔a_{k'i}
14      for i←k+1 to n                   ▷计算 Schur 余子式
15          do a_{ik}←a_{ik}/a_{kk}
16              for j←k+1 to n
17                  do a_{ij}←a_{ij}−a_{ik}a_{kj}
18  LUP-DECOMPOSITION(A,k+1)
```

算法 4-4　对矩阵的 LUP 分解过程

算法 4-4 是一个递归过程，首次调用前须对表示置换矩阵 $\boldsymbol{P}$ 的全局量数组 $\pi[1..n]$ 做初始化。第一次调用时，表示递归层次的参数 k 应传递值 1。第 4～8 行确定在当前需求其 LUP 分解的 $(n-k+1)\times(n-k+1)$ 矩阵首列中具有最大绝对值的元素 $a_{k'k}$。若当前首列的所有元素均为 0，第 9 行和第 10 行报告该矩阵是奇异的。在第 11 行交换 $\pi[k']$ 和 $\pi[k]$ 表示第 k 行与第 k' 行进行了置换，并在第 12 行和第 13 行实际交换 $\boldsymbol{A}$ 的第 k 行与第 k' 行，如此构成枢轴 a_{kk}(整行交换，因为在上述的推导中，不仅是 $\boldsymbol{A}'\boldsymbol{v}\boldsymbol{w}^T/a_{k1}$ 还有 $\boldsymbol{v}/a_{k1}$ 都要与 $\boldsymbol{P}$ 相乘)。最后，在第 14～17 行计算 Schur 余子式，此处写的代码实现了“原地”操作。第 18 行对计算所得的 Schur 余子式进行递归。

由于每次递归，第 2 个参数 k 增加 1，注意第 1 行和第 2 行根据参数 k 是否达到 n 决定递归是否继续。所以一共递归 n 次，每次递归，主要耗时发生在第 14～17 行的双重 **for** 循环的 $O(n^2)$ 次重复上。所以，算法 LUP-DECOMPOSITION 的运行时间为 $O(n^3)$。

图 4-2 所示为 LUP-DECOMPOSITION 的运行。图 4-2(a)为对矩阵 $\boldsymbol{A}$ 的第 1 层操作。

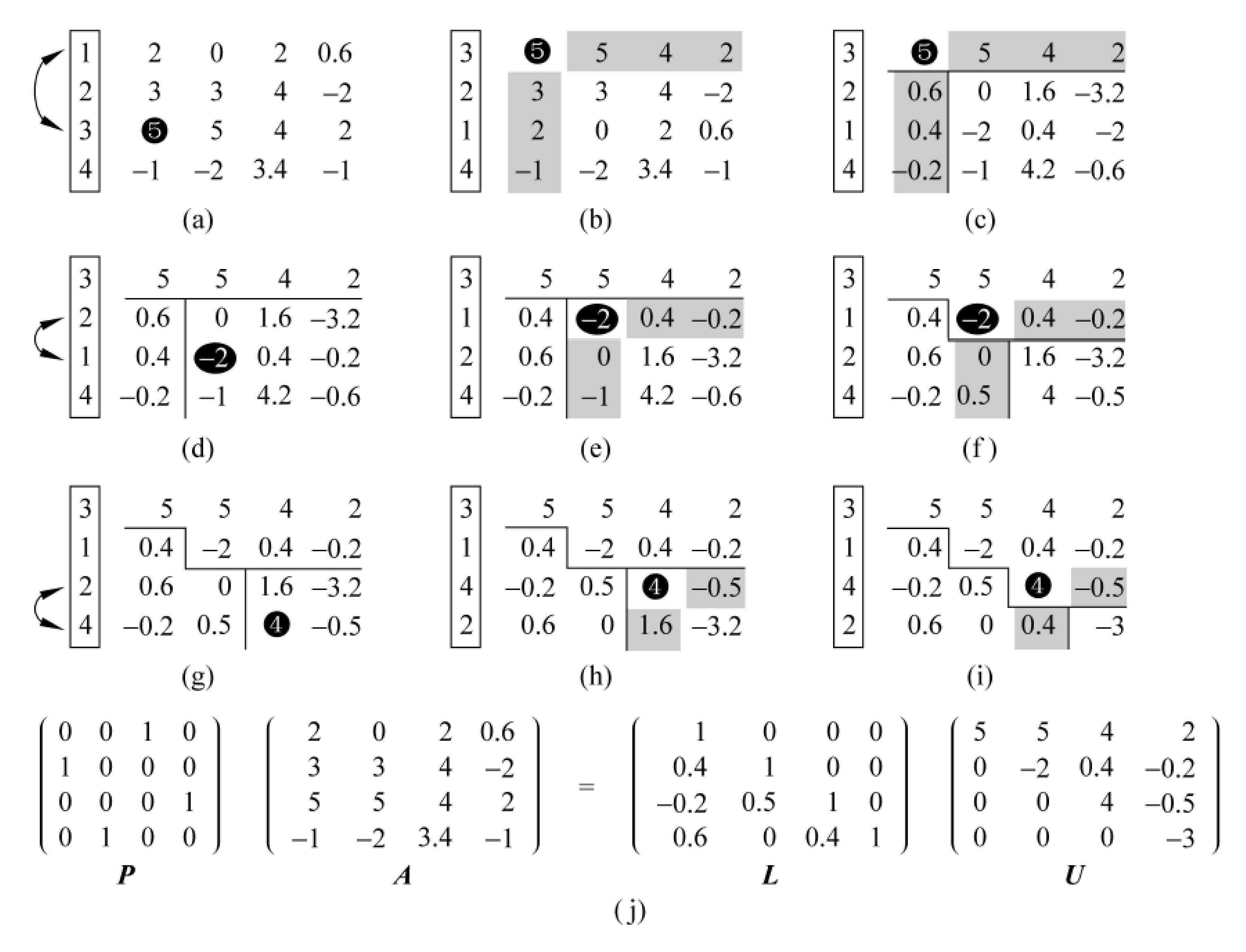


图 4-2　LUP-DECOMPOSITION 的运行

算法的第一步确定首列元素 5 为基准元素,表示在黑圈中。图 4-2(b)为交换第 1 行和第 3 行并更新置换矩阵。阴影列与阴影行表示 $\boldsymbol{v}$ 和 $\boldsymbol{w}^T$。图 4-2(c)为向量 $\boldsymbol{v}$ 替代为 $\boldsymbol{v}/5$,且将矩阵的右下角更新为 Schur 余子式。各条直线将矩阵分割成 3 个区域:$\boldsymbol{U}$ 的元素(上方)、$\boldsymbol{L}$ 的元素(左方),以及 Schur 余子式的元素(右下方)。图 4-2(d)至图 4-2(f)为第 2 层递归。图 4-2(g)至图 4-2(i)为第 3 层递归。第 4 层递归后将不会发生任何变化了,所以是最后一层。图 4-2(j)为所得的 LUP 分解。

算法 4-2 是末尾递归,很容易将其转换为等价的迭代版本。

```
LUP-DECOMPOSITION(A)
1  n←rows[A]
2  for i←1 to n                       ▷初始化置换矩阵 P
3      do π[i]←i
4  for k←1 to n
5      do p←0
6         for i←k to n                ▷搜寻第 k 列下方绝对值最大的元素
7            do if |a_ik|>p
8                  then p←|a_ik|
9                       k'←i
10           if p=0                   ▷若第 k 列下方元素均为 0
11              then error "singular matrix"
12           exchange π[k]↔π[k']      ▷a_k'k 是第 k 列下方元素中绝对值最大者
13           for i←1 to n             ▷交换第 k 行与第 k'行
14              do exchange a_ki↔a_k'i
15           for i←k+1 to n           ▷计算 Schur 余子式
16              do a_ik←a_ik/a_kk
17                 for j←k+1 to n
18                    do a_ij←a_ij−a_ik a_kj
```

算法 4-5 对矩阵的 LUP 分解过程的递归版本

算法 4-5 用一个外层的循环(第 4~18 行的 **for** 循环)替代各层递归。由于循环结构为 3 重嵌套,故运行时间为 $O(n^3)$。

4.2.4 程序实现

下面仅实现 LUP 分解算法的迭代版本。

```
Matrix lupDecomposition(Matrix a,int * pi){
    int i,j,k,k1,n=a.row;
    double p;
    Matrix lu=newMatrixByArry(a.tab,a.row,a.col);
    for(i=0;i<n;i++)
        pi[i]=i;
    for(k=0;k<n;k++){
```

```
8           p=0.0;
9           for(i=k;i<n;i++)
10              if(fabs(lu.tab[i*n+k])>p){
11                  p=fabs(lu.tab[i*n+k]);
12                  k1=i;
13              }
14          assert(p>1e-10);
15          swap(pi+k,pi+k1,sizeof(int));
16          swapRows(lu,k,k1);
17          for(i=k+1;i<n;i++){
18              lu.tab[i*n+k]=lu.tab[i*n+k]/lu.tab[k*n+k];
19              for(j=k+1;j<n;j++)
20                  lu.tab[i*n+j]=lu.tab[i*n+j]-lu.tab[i*n+k]*lu.tab[k*n+j];
21          }
22      }
23      return lu;
24 }
```

程序 4-3　实现对方阵进行 LUP 分解的算法的函数

程序 4-3 的说明如下。

(1) 算法 4-4 和算法 4-5 都将返回两个对象：矩阵 $\boldsymbol{A}$ 的 LU 分解和表示置换矩阵 $\boldsymbol{P}$ 的数组 π。对 C 函数来说，不能返回两个值。为此，将函数 lupDecomposition 的返回值设为 Matrix 型参数 a 的 LU 分解，当然也是 Matrix 型的。而将表示置换矩阵 $\boldsymbol{P}$ 的数组 π，表示成一个指向整型数组的指针参数 pi。

(2) 与算法 4-5 略有不同，lupDecomposition 对矩阵 a 的 LUP 分解并不在 a 的"本地"进行，而是将 a 复制为 Matrix 型变量 lu(第 4 行)，这样做是因为在很多应用中，对矩阵进行 LUP 分解的同时还要保留矩阵本身。对 a 的 LUP 分解在矩阵 lu 上进行，最后函数返回 lu (第 23 行)。

(3) lupDecomposition 函数体内的代码结构与算法伪代码的十分接近，第 14 行调用库函数 assert 保证 $p \neq 0$。用 $p > 10^{-10}$ 这种形式表示 $p \neq 0$，是因为在计算机中浮点数不保证精度以外的数字是否准确，所以浮点型数据需要按一定的精度判断是否等于 0.0 。第 16 行调用函数 swapRows，交换矩阵中指定的两行。该函数定义于文件夹 algebra 中的源文件 matrix.c(原型声明于 matrix.h 中)，读者可打开文件研读。

程序 4-3 中的函数定义于文件夹 algebra 中的源文件 lup.c 中，其原型声明保存在同一文件夹的头文件 lup.h 中。

4.3　解线性方程组

4.3.1　前代法和回代法

对给定的 $\boldsymbol{L}$、$\boldsymbol{P}$ 和 $\boldsymbol{b}$，前代法能在 $\Theta(n^2)$ 时间内解下式(4-12)。注意，我们是用紧凑的数

组形式 $\pi[1..n]$来表示置换矩阵 $\boldsymbol{P}$。对 $i=1,2,\cdots,n,\pi[i]$表示 $P_{i\pi[i]}=1$ 及 $j\neq\pi[i]$时 $P_{ij}=0$。$\boldsymbol{PA}$ 的第 i 行和第 j 列元素为 $a_{\pi[i]j}$,而 $\boldsymbol{Pb}$ 的第 i 个元素为 $b_{\pi[i]}$。由于是单位下三角矩阵,式(4-12)可以重写为

$$\begin{aligned}
&y_1 = b_{\pi[1]}\\
&l_{21}y_1 + y_2 = b_{\pi[2]}\\
&l_{31}y_1 + l_{32}y_2 + y_3 = b_{\pi[3]}\\
&\vdots\\
&l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n = b_{\pi[n]}
\end{aligned}$$

可以直接解得 y_1,因为第一个方程告诉我们 $y_1=b_{\pi[1]}$。解得 y_1,可以将其代入到第二个方程,得到:

$$y_2 = b_{\pi[2]} - l_{21}y_1$$

现在,可以把 y_1 和 y_2 代入到第三个方程,得到:

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2)$$

一般地,把 $y_1,y_2,\cdots,y_{i-1}$“前代入”第 i 个方程解得 y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$$

回代法类似于前代法。给定 $\boldsymbol{U}$ 和 $\boldsymbol{y}$,首先解第 n 个方程,然后向回代入直至第一个方程。如前代法那样,这个过程运行 $\Theta(n^2)$时间。由于 $\boldsymbol{U}$ 是上三角形矩阵,可以重写式(4-13),如

$$\begin{aligned}
&u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n = y_1\\
&u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n = y_2\\
&\vdots\\
&u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n = y_{n-2}\\
&u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = y_{n-1}\\
&u_{n,n}x_n = y_n
\end{aligned}$$

于是,接连解得 $x_n,x_{n-1},\cdots,x_1$ 如下:

$$\begin{aligned}
&x_n = y_n/u_{n,n}\\
&x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}\\
&x_{n-2} = (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2}\\
&\vdots
\end{aligned}$$

或,一般地,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)/u_{ii}$$

给定 $\boldsymbol{P}$、$\boldsymbol{L}$、$\boldsymbol{U}$ 及 $\boldsymbol{b}$,过程 LUP-SOLVER 结合前代法与回代法解 $\boldsymbol{x}$。

LUP-SOLVE(L,U,π,b)

$n \leftarrow rows[L]$

for $i \leftarrow 1$ **to** n

  do $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$

for $i \leftarrow n$ **downto** 1

```
5       do x_i ← (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
6 return x
```

算法 4-6　利用系数矩阵的 LUP 分解解线性方程组的过程

过程 LUP-SOLVE 的第 2 行和第 3 行用前代法解 y，然后在第 4 行和第 5 行用回代法解 x。由于在 **for** 循环的每次求和含有一个隐性的循环，所以运行时间为 $\Theta(n^2)$。

作为本方法的一个例子，考虑如下定义的方程组：

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix}$$

其中

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix}$$

我们希望解出未知数 $\boldsymbol{x}$。LUP 分解是

$$\boldsymbol{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix}, \quad \boldsymbol{U} = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix}, \quad \boldsymbol{P} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

读者可验证 $\boldsymbol{PA}=\boldsymbol{LU}$。用前代法对 $\boldsymbol{Ly}=\boldsymbol{Pb}$ 解 $\boldsymbol{y}$：

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix}$$

通过先算 y_1，然后算 y_2，最后算 y_3，得 $\boldsymbol{y} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$。

用回代法对 $\boldsymbol{Ux}=\boldsymbol{y}$ 解 $\boldsymbol{x}$：

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

所以，得到要求的答案 $\boldsymbol{x} = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$。

4.3.2　用 LUP 分解计算矩阵的逆

假定对矩阵 $\boldsymbol{A}$ 有一个 LUP 分解，即有 3 个矩阵 $\boldsymbol{L}$、$\boldsymbol{U}$ 和 $\boldsymbol{P}$，使得 $\boldsymbol{PA}=\boldsymbol{LU}$。利用 LUP-SOLVER，可以在时间 $\Theta(n^2)$ 内解方程 $\boldsymbol{Ax}=\boldsymbol{b}$。由于 LUP 分解依赖于 $\boldsymbol{A}$ 而与 $\boldsymbol{b}$ 无关，可以用另一个 $\Theta(n^2)$ 时间将 LUP-SOLVER 运行于形如 $\boldsymbol{Ax}=\boldsymbol{b}'$ 的方程。一般地，一旦有了 $\boldsymbol{A}$ 的 LUP 分解，就能用 $\Theta(kn^2)$ 的时间解 k 个版本的方程 $\boldsymbol{Ax}=\boldsymbol{b}$，它们中仅有 $\boldsymbol{b}$ 的不同而已。

方程

$$\boldsymbol{AX} = \boldsymbol{I}_n \tag{4-16}$$

可以视为 n 个不同的形式为 $\boldsymbol{Ax}=\boldsymbol{b}$ 的方程构成的集合。这些方程确定了 $\boldsymbol{A}$ 的逆矩阵 $\boldsymbol{X}$。更准确地说，设 $\boldsymbol{X}_i$ 表示 $\boldsymbol{X}$ 的第 i 列，回忆单位向量 $\boldsymbol{e}_i$ 是 $\boldsymbol{I}_n$ 的第 i 个列。式(4-16)就能因此用对 $\boldsymbol{A}$ 的 LUP 分解，分别对每个方程

$$\boldsymbol{AX}_i = \boldsymbol{e}_i$$

解出 $\boldsymbol{X}_i$，进而解得 $\boldsymbol{X}$。具体的伪代码描述如下。

```
MATRIX-INVERSE(A)
n←row[A]
allocate X as a n×n matrix
(L,U,P)←LUP-DECOMPOSITION(A)
for i←1 to n
    do X_i←LUP-SOLVE(L,U,P,e_i)
return X
```

算法 4-7 利用矩阵的 LUP 分解计算逆矩阵

每个列 $\boldsymbol{X}_i$ 能在 $\Theta(n^2)$时间内求得，所以用对 $\boldsymbol{A}$ 的 LUP 分解在 $\Theta(n^3)$时间内计算出 $\boldsymbol{X}$。由于可以在 $\Theta(n^3)$时间内算得 $\boldsymbol{A}$ 的 LUP 分解，所以可以在时间内 $\Theta(n^3)$确定 $\boldsymbol{A}$ 的逆矩阵 $\boldsymbol{A}^{-1}$。

4.3.3 程序实现

假定已得到系数矩阵 a 的 LUP 分解，可将解线性方程组的算法和计算 a 的逆矩阵的算法实现如下。

```
double* lupSolve(Matrix lu,int *pi,double *b){
    int i,j,n=lu.row;
    double *x=(double*)calloc(n,sizeof(double));
    for(i=0;i<n;i++){
        x[i]=b[pi[i]];
        for(j=0;j<i;j++)
            x[i]=x[i]-lu.tab[i*n+j]*x[j];
    }
    for(i=n-1;i>=0;i--){
        for(j=i+1;j<n;j++)
            x[i]=x[i]-lu.tab[i*n+j]*x[j];
        x[i]=x[i]/lu.tab[i*n+i];
    }
    return x;
}
Matrix matrixInverse(Matrix a){
    double *e=NULL,*xi=NULL;
    int n=a.row,*pi=(int*)calloc(n,sizeof(int)),i;
```

```
19      Matrix b=newMatrix(n,n),lu,x;
20      lu=lup(a,pi);
21      for(i=0;i<n;i++){
22          if(e)free(e);
23          e=(double *)calloc(n,sizeof(double));
24          e[i]=1.0;
25          if(xi)free(xi);
26          xi=lupSolve(lu,pi,e);
27          memcpy(b.tab+i*n,xi,n*sizeof(double));
28      }
29      x=transform(b);
30      clrMatrix(lu);clrMatrix(b);
31      free(pi);free(e);free(xi);
32      return c;
33  }
```

程序 4-4　解线性方程组函数和计算逆矩阵函数

对程序 4-4 的说明如下。

(1) 第 1～15 行定义的函数 lupSolve 实现算法 4-6，解系数矩阵 a 的 LUP 分解为 lu(包含下三角阵 $\boldsymbol{L}$ 和上三角阵 $\boldsymbol{U}$)，pi(表示置换矩阵 $\boldsymbol{P}$ 的数组)的线性方程组 ax=b。其中，第 4～8 行的 **for** 循环实现算法 4-6 中第 2 行和第 3 行用前代法解 y。与伪代码在形式上有所不同：

① 用第 5～7 行实现伪代码中的求和计算 $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$。

② 没有为向量 y 开辟专有的存储空间，而是借用向量 x 的空间存储解 y。这是因为在后面的后代法解 $x[i]$时，只需要用到 $y[i]$和 $x[n-1]$，$x[n-2]$，…，$x[i+1]$的值($i=n-1$，$n-2$，…，0)。所以，不会影响向量 x 的计算。

第 9～12 行的 **for** 循环实现算法 4-6 中第 4 行和第 5 行用后代法计算解向量 x。其中，第 10～12 行实现伪代码中的求和计算 $x_i \leftarrow \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}$。

(2) 第 16～33 行定义的函数 matrixInverse 实现算法 4-7，计算 Matrix 型参数 a 的逆矩阵，并作为函数值返回。其中，第 20 行调用函数 lupDecomposition 计算 a 的 LUP 分解，结果保存在矩阵 lu 和数组 pi 中。第 21～28 的 **for** 循环实现算法 4-7 中第 4 行和第 5 行得 **for** 循环，计算逆矩阵 x 中的每一列 xi。与算法的伪代码相比，形式上有如下区别。

① 第 22～24 行设置第 i 个分量为 1，其他分量为 0 的单位向量 ei。

② 第 25 行和第 26 行调用 lupSolve 计算解方程 ax=ei，计算逆矩阵的第 i 列 xi。

③ 由于定义的矩阵中数表是以行优先原则存储的，所以第 26 行将 xi 暂时存于矩阵 b 的第 i 行。

第 29 行调用函数 transform，计算矩阵 b 的转置得出 a 的逆矩阵 x。在 32 行返回 x 之前，第 30 行释放 Matrix 型变量 lu、b 的存储空间，第 31 行释放向量 e、pi、xi 的存储空间以防内存泄漏。

程序 4-4 定义的函数保存于文件夹 algebra 中的源文件 lup.c(原型声明于头文件 lup.

h)中,以备调用。

4.4 多项式及其计算

4.4.1 多项式及其表示

变量 x 取自于一个代数域 F 的多项式是一个表示成和式的函数:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j。$$

把 a_0、a_1、…、a_{n-1}称为该多项式的系数。系数取自于域 F,通常是复数域 C。多项式 $A(x)$称为是 k **次**的,若其最高非零系数是 a_k。任何大于多项式次数的整数称为该多项式的**次界**。因此,以 n 为次界的多项式次数可能是 $0 \sim n-1$ 之间的任何一个整数。

1. 系数表示法

以 n 为次界的多项式 $A(x) = \sum_{j=0}^{n-1} a_j x^j$ 的系数表示法是一个由系数构成的向量 $\boldsymbol{a} = \langle a_0, a_1, \cdots, a_{n-1} \rangle$。

系数表示法对多项式的一些运算来说是方便的,例如,对多项式 $A(x)$在给定点 x_0 处的**求值**计算。利用 **Horner 规则**,求值:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))))$$

下列算法实现 Horner 规则。

```
HORNER(a0,…,an-1,x0)
1 y0←0
2 for i ← n-1 downto 0
3       do y0← ai+x0 · y0
4 return y0
```

算法 4-8 计算多项式值的 Horner 算法

算法 4-8 耗时 $\Theta(n)$。这是一个渐增型算法,第 2～4 行 **for** 循环的一个循环不变量叙述如下。

在第 2～4 行的 **for** 循环的每次重复之初,

$$y_0 = \sum_{k=0}^{n-(i+1)} a_{k+i} x_0^k$$

为证明这个循环不变量,考察第一次重复之初,$y_0 = a_{n-1}$,$i = n-1$。$\sum_{k=0}^{n-(i+1)} a_{k+i} x_0^k = a_{n-1} = y_0$。这说明此时循环不变量为真。设 $1 < i < n$ 时循环不变量为真,即本次重复之初 $y_0 = \sum_{k=0}^{n-(i+1)} a_{k+i} x_0^k$。在本次重复中,第 3 行执行 $y_0 \leftarrow a_i + x_0 \cdot y_0$。这蕴涵着

$$y_0 = a_i + x_0\left(\sum_{k=0}^{n-(i+1)} a_{k+i}x_0^k\right) = \sum_{k=0}^{n-i} a_{k+i}x_0^k$$

则执行 $i \leftarrow i-1$，则上式变成 $\sum_{k=0}^{n-(i+1)} a_{k+i}x_0^k$，且此状态将维持到下一次重复之初。这就证明了次循环不变量是正确的。循环终止时，$i=-1$，$y_0=\sum_{k=0}^{n} a_k x_0^k$。这恰是想要得到的。

2. 点值表示法

以 n 为次界的多项式 $A(x)$ 的点值表示法是 n 个点值对构成的集合

$$\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$$

使得所有的 x_k 都不相同且

$$y_k = A(x_k) \quad k = 0, 1, \cdots, n-1 \tag{4-17}$$

一个多项式有很多点值表达形式，这是因为任意 n 个不同的点 $x_0, x_1, \cdots, x_{n-1}$ 都能用来构成一个点值表示。

3. 系数表示与点值表示的转换法

对一个系数法表示的多项式计算它的一个点值表示形式是很直接的，这是因为只要选取 n 个不同的点 x_0、x_1、$\cdots$、x_{n-1} 并计算 $A(x_k)$，$k=0,1,\cdots,n-1$。用 Horner 方法，这 n 个求值计算耗时 $\Theta(n^2)$。稍后会看到，若巧妙地选取 x_k，这一计算可以加速为运行时间是 $\Theta(n\lg n)$。

求值计算的逆——由多项式的一个点值表达形式确定其系数表达形式——称为**插值**。定理 4-5 说明只要假定欲插值的多项式的次界等于给定的点值对数，插值定义是良好的。

定理 4-5（多项式插值的唯一性）　对于任意的 n 个点值对集合 $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$，若所有的 x_k 各不相同，则存在唯一的以 n 为次界的多项式 $A(x)$，满足 $y_k = A(x_k)$，$k=0,1,\cdots,n-1$。

证明　此证明是基于矩阵的逆的存在性的。式(4-17)等价于矩阵方程

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{4-18}$$

左边的矩阵记为 $V(x_0, x_1, \cdots, x_{n-1})$ 并称为 Vander-monde 矩阵。根据线性代数，此矩阵的行列式是

$$\prod_{0 \leqslant j < k \leqslant n-1} (x_k - x_j)$$

若 x_k 各不相同，根据定理 4-4，它是可逆的。于是，各系数 a_j 可以被点值表达形式唯一解得

$$a = V(x_0, x_1, \cdots, x_{n-1})^{-1} y$$

定理 4-5 的证明描述了基于解线性方程组(4-18)的插值算法。利用 4.2 节的 LUP 分解算法，可以在 $O(n^3)$ 时间内解出它来。

最后，来考虑如何对一个表示为点值形式的多项式求在一个新的点处的值。这可先将点值形式转换成系数形式，然后对系数形式求在新点处的值。

4.4.2 多项式的运算

1. 系数形式

人们希望对多项式定义一些运算。对**多项式加法**,若 $A(x)$和 $B(x)$是以 n 为次界的多项式,说它们的**和** $C(x)$也是以 n 为次界一个多项式,使得对域中所有的 x 均有 $C(x)=A(x)+B(x)$。也就是说,若

$$A(x)=\sum_{j=0}^{n-1}a_jx^j \quad 及 \quad B(x)=\sum_{j=0}^{n-1}b_jx^j$$

则

$$C(x)=\sum_{j=0}^{n-1}c_jx^j$$

其中 $c_j=a_j+b_j, j=0,1,\cdots,n-1$。例如,若有多项式 $A(x)=6x^3+7x^2-10x+9$ 及 $B(x)=-2x^3+4x-5$,则 $C(x)=4x^3+7x^2-6x+4$。由系数向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$和 $\boldsymbol{b}=(b_0,b_1,\cdots,b_{n-1})$表示的两个次界为 n 的多项式 A 和 B 的加法表示成伪代码过程如下。

```
SUM(A,B)
for j←0 to n-1
    do c_j←a_j+b_j
return C              ▷C 是系数向量,为< c_0,c_1,…,c_{n-1} >的多项式
```

算法 4-9 系数形式多项式的加法过程

不难看出,SUM 过程耗时 $\Theta(n)$。

对**多项式乘法**,若 $A(x)$和 $B(x)$是以 n 为次界的多项式,它们的**积** $C(x)$是以 $2n-1$ 为次界一个多项式,使得对域中所有的 x 均有 $C(x)=A(x)\ B(x)$。你可能曾经通过把 $A(x)$的每个项与 $B(x)$的每个项相乘,然后合并同类项来将两个多项式相乘。例如,可将 $A(x)=6x^3+7x^2-10x+9$ 和 $B(x)=-2x^3+4x-5$ 相乘:

$$\begin{array}{r}
6x^3+7x^2-10x+9 \\
\times)\quad -2x^3 \qquad\quad +4x-5 \\
\hline
-30x^3-35x^2+50x-45 \\
24x^4+28x^3-40x^2+36x \quad \\
+)\ -12x^6-14x^5+20x^4-18x^3 \qquad\qquad\qquad \\
\hline
-12x^6-14x^5+44x^4-20x^3-75x^2+86x-45
\end{array}$$

另一种表示积 $C(x)$的方式是

$$C(x)=\sum_{j=0}^{2n-2}c_jx^j \tag{4-19}$$

其中

$$c_j=\sum_{k=0}^{j}a_kb_{j-k} \tag{4-20}$$

注意 $\text{degree}(C)=\text{degree}(A)+\text{degree}(B)$。

将式(4-19)和式(4-20)写成伪代码过程如下。

```
PRODUCT(A,B)
for j ← 0 to 2n−2
    do c_j ← 0
        for k ← 0 to j
            do c_j ← c_j + a_k b_{j−k}
return C
```

算法 4-10　系数形式多项式乘法过程

用过程 PRODUCT 计算多项式乘积将耗时 $\Theta(n^2)$，这是因为向量 $\boldsymbol{a}$ 中的每一个系数必须与向量 $\boldsymbol{b}$ 中的每一个系数相乘。系数表示的多项式的乘法看来比求值运算或多项式加法运算要难得多。由式(4-20)给出的结果系数向量 $\boldsymbol{c}$ 也称为输入向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的**卷积**，记为 $\boldsymbol{c}=\boldsymbol{a}\otimes\boldsymbol{b}$。由此可见，如何更快地计算向量卷积是有效计算多项式乘积的基本因素。

2. 点值表示

对多项式的很多运算点值表达形式都是很方便的。对加法而言，若 $C(x)=A(x)+B(x)$，则对任一点 x_k，$C(x_k)=A(x_k)+B(x_k)$。若有 A 的一个点值表达形式：

$$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})\}$$

和 B 的点值表达形式：

$$\{(x_0,y'_0),(x_1,y'_1),\cdots,(x_{n-1},y'_{n-1})\}$$

注意 A 和 B 在相同的 n 个点处求值，则 C 的一个点值表达形式为

$$\{(x_0,y_0+y'_0),(x_1,y_1+y'_1),\cdots,(x_{n-1},y_{n-1}+y'_{n-1})\}$$

于是，将两个以 n 为次界的多项式的点值表达形式相加耗时 $\Theta(n)$。类似地，点值表示形式对多项式的乘法也是很方便的。若 $C(x)=A(x)B(x)$，则对任一点 x_k，$C(x_k)=A(x_k)B(x_k)$。且能按对应点相乘来把 A 的点值表达式与 B 的点值表达式相乘得到 C 的点值表达式。

必须面对 C 的次界是 A 的次界与 B 的次界之和。A 和 B 的标准点值表达形式分别由 n 个点值对构成。将它们相乘得到 C 的点值表达形式，但由于 C 的次界是 $2n$，需要 $2n$ 个点值对作为 C 的点值表达式。所以必须从“扩展”A 和 B 的点值表达式为含有 $2n$ 个点值对开始。给定 A 的扩展点值表达式：

$$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{2n-1},y_{2n-1})\}$$

以及 B 的扩展点值表达式：

$$\{(x_0,y'_0),(x_1,y'_1),\cdots,(x_{2n-1},y'_{2n-1})\}$$

则 C 的点值表达式为

$$\{(x_0,y_0y'_0),(x_1,y_1y'_1),\cdots,(x_{2n-1},y_{2n-1}y'_{2n-1})\}$$

给定两个扩展点值表示形式的多项式作为输入，将它们相乘得到结果的点值表达式的时间是 Θn，这要比将两个系数形式相乘所需的时间少得多。

4.4.3　FFT

能使用线性时间的点值形式多项式乘法来加速系数形式的多项式乘法吗？答案取决于

从系数形式转换成点值形式(求值)和反过来(插值)的能力。

可以选取任何点作为求值点，通过仔细选择求值点，可以仅用 $\Theta(n\lg n)$ 时间在两种表示形式之间转换。如将在本节中看到的，若选择“复单位根”作为求值点，可以通过计算系数向量的“离散 Fourier 变换”(Discrete Fourier Transform 或 DFT)，产生一个点值形式。其逆变换、插值，可以通过对点值形式取“DFT 的逆”而得到系数向量。本节将说明 FFT 是如何在 $\Theta(n\lg n)$ 时间内执行 DFT 及 DFT 的逆的。

图 4-3 形象地说明了此策略。要注意次界。两个以 n 为次界的多项式的积的次界是 $2n$。在对输入的多项式 A 和 B 求值之前，要先通过增加 n 个系数为 0 的高次数项来将它们的次界增加到 $2n$。由于向量有 $2n$ 个元素，用“$2n$ 次单位根”，在图 4-3 中表为 ω_{2n}。

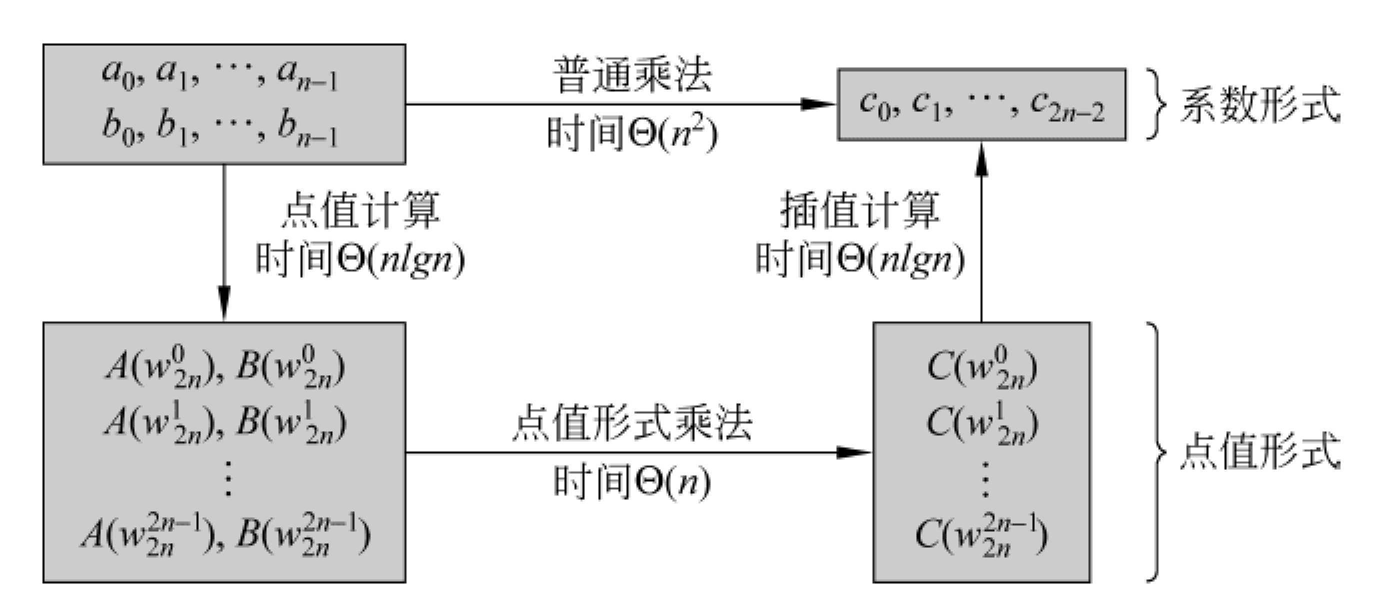


图 4-3　一种有效的多项式乘法的图形轮廓

图 4-3 所示为一种有效的多项式乘法的图形轮廓。顶部的表示形式是系数形式，底部的是点值表达形式。自左向右的箭头对应于乘法运算。ω_{2n} 表示 $2n$ 次单位根。

给定 FFT，有以下的 $\Theta(n\lg n)$ 时间对次界为 n 的多项式 $A(x)$ 和 $B(x)$ 的乘法过程，其中输入与输出都是系数形式。假定 n 是 2 的整幂；这一要求总可以通过添加系数为 0 的高次项而得到满足。

(1) 两倍次界：通过对 $A(x)$ 和 $B(x)$ 的系数表达式中创建系数为 0 的高次项来完成。

(2) 求值：通过两次应用 $2n$ 阶 FFT，求得 $A(x)$ 及 $B(x)$ 点值表达式。这两个表达式包含了两个多项式在 $2n$ 次单位根处的值。

(3) 按点相乘：通过将这些值的按点相乘，计算多项式 $C(x)=A(x)B(x)$ 的点值表达式。此表达式包含了 $C(x)$ 在 $2n$ 次单位根处的值。

(4) 插值：通过对 $2n$ 个点值对调用 FFT 计算 DFT 的逆计算多项式 $C(x)$ 的系数表达式。

步骤(1)和(3)耗时 $\Theta(n)$，步骤(2)和(4)耗时 $\Theta(n\lg n)$。因此，一旦说明了如何使用 FFT，将证明定理 4-6。

定理 4-6　输入与输出均为系数形式，两个以 n 为次界的多项式的积可以在 $\Theta(n\lg n)$ 时间内算得。

1. DFT 和 FFT

刚才声称若使用复单位根，就能在 $\Theta(n\lg n)$ 时间内进行求值和插值。下面，就来定义复单位根并研究它们的性质，定义 DFT，说明 FFT 如何在 $\Theta(n\lg n)$ 时间内计算 DFT 及其逆。

1）**复单位根**

一个 n 次复单位根是一个复数 ω，使得 $\omega^n=1$。

恰有 n 个 n 次复单位根，它们是 $e^{2\pi ik/n}$，$k=0,1,\cdots,n-1$。为解释这一结论，利用复数的指数形式定义：

$$e^{iu} = \cos(u) + i\sin(u)$$

图 4-4 展示了 n 个复单位根分布在复平面中，并以原点为圆心，半径为 1 的圆上。值

$$\omega_n = e^{-2\pi i/n} \tag{4-21}$$

称为 n 次复单位主根，其他的 n 次复单位根都是 ω_n 的幂，即 n 个 n 次复单位根是

$$\omega_n{}^0, \omega_n{}^1, \cdots, \omega_n{}^{n-1}$$

n 次复单位根的基本性质由如下引理给出：

$$\omega_n = e^{-2\pi i/n}$$

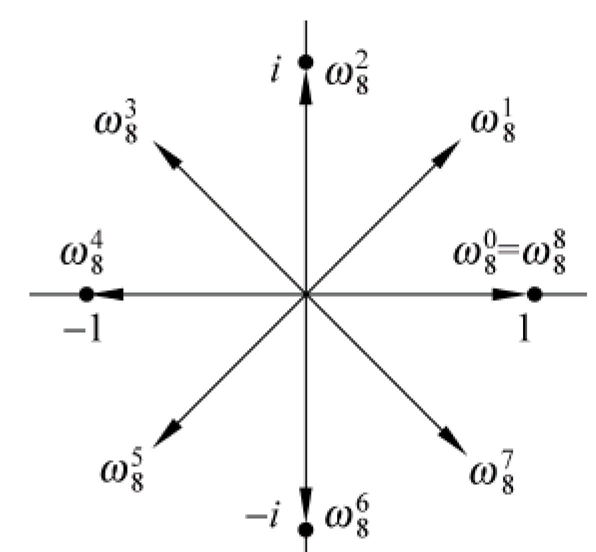


图 4-4　复平面中的值 $\omega_8{}^0, \omega_8{}^1, \cdots, \omega_8{}^7$

（其中 $\omega_8=e^{2\pi ik/8}$ 是 8 次复单位主根）

引理 4-1（消去引理）　对任一整数 $n\geqslant 0$，$k\geqslant 0$，及 $d>0$：

$$\omega_{dn}^{dk} = \omega_n^k \tag{4-22}$$

证明　本引理直接由式(4-21)而得，这是因为

$$\omega_{dn}{}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n{}^k$$

推论 4-2　对任一偶数 $n>0$，

$$\omega_n^{n/2} = \omega_2 = -1$$

证明　这是因为

$$\begin{aligned} \omega_n{}^{n/2} &= \omega_{2\cdot n/2}{}^{1\cdot n/2} && \text{等值变换} \\ &= \omega_2{}^1 && \text{消去律} \\ &= \omega_2 \\ &= -1 && \text{式(4-21)} \end{aligned}$$

引理 4-2（折半引理）　若 $n>0$ 是偶数，则 n 个 n 次复单位根的平方是 $n/2$ 个 $n/2$ 次复单位根。

证明　根据消去引理，对任一非负整数 k，有 $(\omega_n{}^k)^2=\omega_{n/2}{}^k$。若对所有 n 个 n 次复单位根进行平方，则每个 $n/2$ 次单位根将得到两次，这是因为

$$\begin{aligned} (\omega_n{}^{k+n/2})^2 &= \omega_n{}^{2k+n} \\ &= \omega_n{}^{2k}\omega_n{}^n \\ &= \omega_n{}^{2k} \\ &= (\omega_n{}^2)^k \end{aligned}$$

于是，ω_n^k 和 $\omega_n^{k+n/2}$ 具有相同的平方值。

正如将会看到的那样，折半引理是多项式的系数形式及点值形式之间进行转换的分治方案的基础。因为，它保证了递归子问题只有一半大。

引理 4-3　对任意整数 $n\geqslant 1$ 及不能被 n 整除的非负整数 k：

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

证明　将 $\omega_n{}^k$ 视为公比 q,则有

$$\begin{aligned}\sum_{j=0}^{n-1}(\omega_n^k)^j &= \sum_{j=0}^{n-1}q^j \\ &= \frac{q^n-1}{q-1} \qquad \text{等比级数前 } n \text{ 项和} \\ &= \frac{(\omega_n^k)^n-1}{\omega_n^k-1} \\ &= \frac{(\omega_n^n)^k-1}{\omega_n^k-1} \\ &= \frac{(1)^k-1}{\omega_n^k-1} \qquad n \text{ 次复单位主根} \\ &= 0\end{aligned}$$

要求 k 不能被 n 整除是为了保证分母不为 0。

2) **DFT**

回顾我们所希望的是对次界为 n 的多项式

$$A(x) = \sum_{j=0}^{n-1}a_j x^j$$

求在 $\omega_n{}^0,\omega_n{}^1,\cdots,\omega_n{}^{n-1}$(即在 n 个 n 次复单位根)[①]处的值。不失一般性,假定 n 是 2 的幂,这是因为对给定的次界总能重新放大——总能添加所需的系数为零的高次项。假定 A 由系数形式给出:$\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$。对 $k=0,1,\cdots,n-1$,定义结果:

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1}a_j\omega_n^{kj} \tag{4-23}$$

向量 $\boldsymbol{y}=(y_0,y_1,\cdots,y_{n-1})$是系数形式 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$的**离散 Fourier 变换(DFT)**。我们写成 $\boldsymbol{y}=\mathrm{DFT}_n(\boldsymbol{a})$。

3) **FFT**

利用称为**快速 Fourier 转换(FFT)**的方法,可以在 $\Theta(n\lg n)$时间内计算 $\mathrm{DFT}_n(\boldsymbol{a})$。该方法利用了复单位根的特性,运行效率高于 $\Theta(n^2)$时间的直接方法。

FFT 方法采用分治策略,将 $A(x)$的偶下标和奇下标系数分别定义两个次界为 $n/2$ 的多项式 $A^{[0]}(x)$和 $A^{[1]}(x)$:

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$

$A^{[0]}$包含 A 的所有下标为偶数的系数(下标的二进制表示以 0 结尾),而 $A^{[1]}$包含 A 的所有下标为奇数的系数(下标的二进制表示以 1 结尾)。这意味着

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{4-24}$$

所以在 $\omega_n^0,\omega_n^1,\omega_n^2,\cdots,\omega_n^{n-1}$ 处的求值约简为如下。

(1) 求次界为 $n/2$ 的多项式 $A^{[0]}(x)$ 和 $A^{[1]}(x)$在点

$$(\omega_n^0)^2,(\omega_n^1)^2,\cdots,(\omega_n^{n-1})^2 \tag{4-25}$$

① 长度 n 在 4.4.1 节中为 $2n$,这是因为在求值之前需要将给定多项式的次界加倍。在多项式乘法讨论中,总是对 $2n$ 次复单位根进行操作。

处的值。

(2) 按式(4-24)合并结果。

根据折半引理，式(4-25)恰含 $n/2$ 个不同的 $n/2$ 次复单位根，每个根在其中恰发生两次。所以，次界为 $n/2$ 的多项式递归地求 $n/2$ 个 $n/2$ 次复单位根处的值。这些子问题的形式与原问题的相同，但规模仅为其半。这样就成功地将 n 个元素的 DFT_n 计算划分成两个 $n/2$ 个元素的 $DFT_{n/2}$ 计算。这一分解是下列的递归 FFT 算法的基础，该算法计算具有 n 个元素的向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$ 的 DFT，其中，n 是 2 的幂。

```
RECURSIVE-FFT(a)
1  n ← length[a]                         ▷n 是 2 的幂
2  if n=1
3      then return a
4  ωn ← e^(2πi/n)
5  ω ← 1
6  a[0] ← (a0, a2, …, an-2)
7  a[1] ← (a1, a3, …, an-1)
8  y[0] ← RECURSIVE-FFT(a[0])
9  y[1] ← RECURSIVE-FFT(a[1])
10 for k ← 0 to n/2-1
11     do yk ← yk[0] + ω yk[1]
12         yk+(n/2) ← yk[0] - ω yk[1]
13         ω ← ω ωn
14 return y                              ▷假定 y 是列向量
```

算法 4-11　计算多项式的 FFT 的递归算法

RECURSIVE-FFT 过程运行如下。第 2 行和第 3 行表示递归头；一个元素的 DFT 是其本身，这是因为在此情形中：

$$y_0 = a_0\omega_1{}^0 = a_0 \cdot 1 = a_0$$

第 6 行和第 7 行定义 $A^{[0]}$ 和 $A^{[1]}$ 的系数。第 4 行、第 5 行及 13 行保证 ω 得以及时更新，使得在第 11 行和第 12 行执行时，$\omega=\omega_n{}^k$(跟踪 ω 的每一次重复的运行值，而不是在每次重复中用一个 **for** 循环来计算 $\omega_n{}^k$，以此节省时间)。第 8 行和第 9 行执行的 $DFT_{n/2}$ 递归计算，其中，对 $k=0,1,\cdots,n/2-1$，置

$$y_k{}^{[0]} = A^{[0]}(\omega_{n/2}{}^k),$$
$$y_k{}^{[1]} = A^{[1]}(\omega_{n/2}{}^k),$$

或，根据消去引理，由于 $\omega_{n/2}{}^k=\omega_n{}^{2k}$

$$y_k{}^{[0]} = A^{[0]}(\omega_n{}^{2k})$$
$$y_k{}^{[1]} = A^{[1]}(\omega_n{}^{2k})$$

第 11 行和第 12 行将两个递归 $DFT_{n/2}$ 计算的结果加以合并。对 $y_0,y_1,\cdots,y_{n/2-1}$，第 11 行得出

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \qquad (\text{根据式}(4\text{-}24)) \end{aligned}$$

其中的最后一个等式是直接遵循式(4-24)。对 $y_{n/2},y_{n/2+1},\cdots,y_{n-1}$，设 $k=0,1,\cdots,n/2-1$，

第 12 行得出

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && (\text{因为 } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && (\text{因为 } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) && (\text{根据式(4-24)}) \end{aligned}$$

于是,由 RECURSIVE-FFT 返回的向量 y 确实是输入 a 的 DFT。

在第 10~13 行的 **for** 循环中,每一个 $y_k^{[1]}$ 值都要与 ω_n^k 相乘,$k=0,1,\cdots,n/2-1$。该积既要与 $y_k^{[0]}$ 相加也要与其相减。因为每个 ω_n^k 因子既要用到正的也要用到负的,因子 ω_n^k 称为**旋转因子**。

为了确定过程 RECURSIVE-FFT 的运行时间,除了递归调用以外,每次耗时 $\Theta(n)$,其中 n 是输入向量的长度。所以运行时间的递归方程为

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n)$$

于是,能在时间 $\Theta(n\lg n)$ 内利用快速 Fourier 变换求得次界为 n 的多项式在 n 次复单位根处的值。

2. 复单位根处的插值

现在通过说明如何将复单位根插入成一个多项式,从而将点值形式转换成系数形式来完成多项式的乘法。

由式(4-18),可以把 DFT 写成矩阵积 $\mathbf{y}=\mathbf{V}_n\mathbf{a}$,其中 $\mathbf{V}_n$ 是由 ω_n 的各次幂构成的矩阵:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^8 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

$\mathbf{V}_n$ 的(k,j)项是 ω_n^{kj};$j,k=0,1,\cdots,n-1$,且 $\mathbf{V}_n$ 的项的指数构成一个乘法表。

对逆运算,写成 $\mathbf{a}=\mathrm{DFT}_n^{-1}(\mathbf{y})$,可以通过将 $\mathbf{y}$ 乘以 $\mathbf{V}_n$ 的逆矩阵 $\mathbf{V}_n^{-1}$ 得到。

定理 4-7 对 $j,k=0,1,\cdots,n-1$,$\mathbf{V}_n^{-1}$的第(j,k)项是 ω_n^{-kj}/n。

证明 $\mathbf{V}_n\mathbf{V}_n^{-1}=\mathbf{I}_n$,即 $n\times n$ 单位阵。考虑 $\mathbf{V}_n\mathbf{V}_n^{-1}$的第$(j,j')$项:

$$\begin{aligned} [\mathbf{V}_n^{-1}\mathbf{V}_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n \end{aligned}$$

根据和式引理(引理 4-12),此和式当 $j'=j$ 时等于 1,否则的话为 0。注意,要求 $-(n-1)<j'-j<n-l$,这样 $j'-j$ 就不会被整除,因此可以应用和式引理。

给定逆矩阵 $\mathbf{V}_n^{-1}$,我们有由式(4-26)给出的 $\mathrm{DFT}_n^{-1}(\mathbf{y})$:

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k\omega_n^{-kj} \tag{4-26}$$

$j=0,1,\cdots,n-1$。比较式(4-23)和式(4-26)可看到，只要把 FFT 算法中交换 $\boldsymbol{a}$ 和 $\boldsymbol{y}$ 的角色，把 ω_n 换成 $\omega_n{}^{-1}$，并把结果中的每个元素除以 n，就算得 DFT 的逆。于是，$\mathrm{DFT}_n{}^{-1}$ 也能在 $\Theta(n\lg n)$时间内算得。

这样，利用 FFT 及 FFT 的逆，能在次界为 n 的多项式的系数形式与点值形式之间用时间来回转换。对多项式的乘法问题，已经证明了定理 4-8。

定理 4-8　对任意的两个长度为 n 的向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，其中 n 是 2 的幂：

$$\boldsymbol{a}\otimes\boldsymbol{b}=\mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(\boldsymbol{a})\cdot\mathrm{DFT}_{2n}(\boldsymbol{b}))$$

其中向量 a 和 b 的长度用 0 填充成为 $2n$，且“·”表示两个长度为 $2n$ 的向量按分量的积。

3. FFT 的迭代实现

RECURSIVE-FFT 的第 10～13 行的 **for** 循环涉及两次对值 $\omega_n^k y_k^{[1]}$ 的计算，这样的值称为**公共子式**。可以改变该循环，将此值保存在临时变量 t 中，使之对其只计算一次。

```
for k←0 to n/2－1
    do t←ωy_k^[1]
       y_k←y_k^[0]＋t
       y_{k+(n/2)}←y_k^[0]－t
       ω←ωω_n
```

在此循环中，用 $y_k{}^{[1]}$乘以旋转因子 $\omega=\omega_n^k$，将积保存于 t 中，并加入 $y_k{}^{[0]}$ 和从 $y_k{}^{[0]}$ 中间去 t，此称为**蝶形运算**。

图 4-5 所示为蝶形运算。图 4-5(a)为两个输入从左边进入，旋转因子 ω_n^k 乘以 $y_k{}^{[1]}$，和与差在右边输出。图 4-5(b)为蝶形运算的简化图。

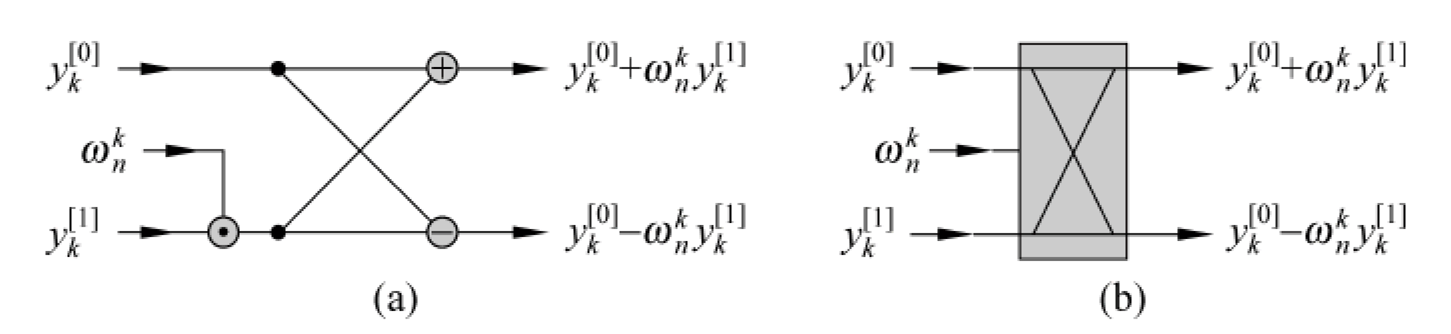


图 4-5　蝶形运算

现在来说明如何将 FFT 算法在结构上从递归改为迭代。在图 4-6 中，已经将初始 $n=8$ 调用RECURSIVE-FFT 时的输入向量表示成树状结构。树中的每个结点对应一次递归调用的输入向量。除非输入向量仅有一个元素，RECURSIVE-FFT 的每次调用都将引发两个递归调用。把第一个递归调用表为左孩子，第二次调用表为右孩子。

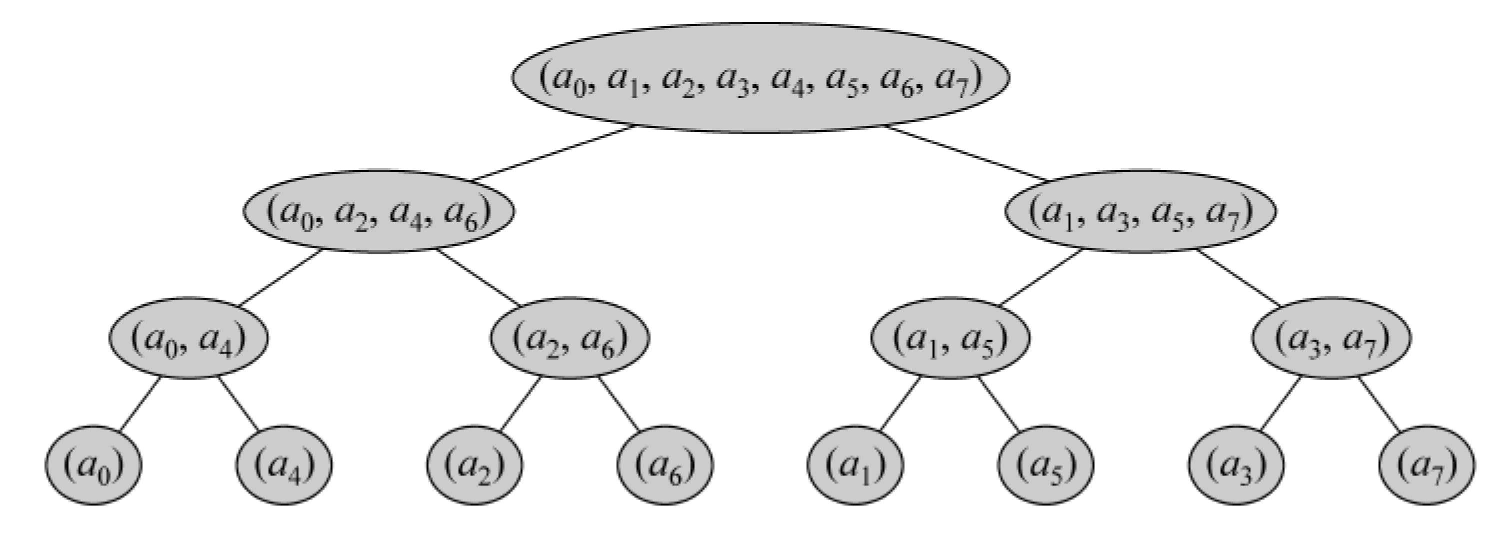


图 4-6　RECURSIVE-FFT 过程递归调用的输入向量树(首次调用 $n=8$)

观察这棵树,发现若将初始向量 **a** 中的元素按它们在树叶中出现的顺序排列,就可以按下列的方式执行 RECURSIVE-FFT 过程。首先,按对取元素,每一对用一个蝶形算子计算它们的 DFT 并取而代之。于是,原向量就包含了 $n/2$ 个双元素的 DFT。接着,对这 $n/2$ 个按对取之,用由两个碟形算子计算的四元素 DFT 替代原来的两个双元素 DFT。于是原向量就包含了 $n/4$ 个四元素 DFT。继续以此形式操作,直至向量含有两个 $n/2$ 元素的 DFT,这样就可以用 $n/2$ 个碟形算子算得最终的 n 元素 DFT。

为将此观察转换成代码,使用一个数组 $A[0..n-1]$,它最初将输入向量 **a** 的元素按它们在图 4-6 的树中叶子的顺序排列存储(稍后将说明如何用一种称为比特翻转的方法来确定这一顺序)。由于在该树的每一层做合并,所以引进一个变量 s 来对层次计数,从 1 开始(在底层,将两个元素合并为双元素 DFT)到 $\lg n$(在顶层,合并两个 $n/2$ 元素的 DFT 得到最终结果),于是算法有如下的结构:

```
1 for s ← 1 to lg n
2   do for k ← 0 to n−1 by 2^s
3      do combine the two 2^(s−1)-element DFT's in
          A[k ..k+2^(s−1)−1] and A[k+2^(s−1)..k+2^s−1]
          into one 2^s-element DFT in A[k ..k+2^s−1]
```

可以将循环体(第 3 行的)表达成更准确的伪代码。从过程 RECURSIVE-FFT 中复制 **for** 循环,用 $A[k .. k+2^{s-1}-1]$替代 $y^{[0]}$,用 $A[k+2^{s-1}.. k+2^s-1]$替代 $y^{[1]}$。每个蝶形算子按不同的 s 值使用旋转因子。旋转因子是 ω_m 的幂,而 $m=2^s$(引入变量 m 是为加强可读性)引入另一个临时变量 u,使得能就地进行蝶形运算。在把第 3 行替换为该循环体后,得到如下的伪代码,它是稍后将介绍的并行实现的基础。此代码首先调用辅助过程 BIT-REVERSE-COPY(a,A),将向量 a 按初始所需的值的顺序复制到数组 A 中。

```
ITERATIVE-FFT(a)
BIT-REVERSE-COPY (a,A)
n ← length[a]                    ▷ n is a power of 2
for s ← 1 to lg n
    do m ← 2^s
       ω_m ← e^(2πi/m)
       for k ← 0 to n−1 by m
           do ω ← 1
              for j ← 0 to m/2−1
                  do t ← ωA[k+j+m/2]
                     u ← A[k+j]
                     A[k+j] ← u+t
                     A[k+j+m/2] ← u−t
                     ω ← ωω_m
```

算法 4-12 计算多项式 FFT 的迭代算法

BIT-REVERSE-COPY 是如何将输入向量 a 中的元素按所需顺序复制到数组 A 中去的呢?图 4-6 的树中叶子顺序称为**比特翻转置换**,即若设 $\text{rev}(k)$为将的二进制表达式翻转后所得的 $\lg n$ 位二进制表达式,则希望把向量元素 a_k 置于数组 $A[\text{rev}(k)]$的位置上。例如,

图 4-6 中叶子排列顺序是 0,4,2,6,1,5,3,7;该序列的二进制表达式为 000,100,010,110,001,101,011,111,当将它们的比特翻转后得到序列 000,001,010,011,100,101,110,111。为理解一般情况下为什么要做比特翻转置换,注意在树的顶部元素下标的最低位为零的都置于左子树中,而下标的最低位为 1 的元素都置于右子树中。在每一层中剥离最低位,继续往下操作,直至叶子层上得到比特翻转置换顺序。

完成整数 k 的二进制比特翻转的函数 $rev(k)$很容易建立,过程如下。

```
BIT-REVERSE-COPY(a,A)
n ← length[a]
for k ← 0 to n−1
    do c←rev(k)
        A[c] ← a_k
```

算法 4-13　计算多项式系数比特反转置换的算法

FFT 的迭代实现的运行时间是 $\Theta(n\lg n)$。BIT-REVERSE-COPY(a,A)过程的调用显然耗时 $O(n\lg n)$,这是因为做 n 次迭代,且用 $O(\lg n)$时间把介于 $0 \sim n-1$ 之间的 $\lg n$ 位二进制整数做比特翻转。为完成 ITERATIVE-FFT 运行于 $\Theta(n\lg n)$时间内的证明,来说明最里层的循环体(第 8～13 行)执行次数 $L(n)$是 $\Theta(n\lg n)$。第 6～13 行的 **for** 循环对每个 s 值重复 $n/m=n/2^s$ 次,而最里层第 8～13 行的循环重复 $m/2=2^{s-1}$次。于是:

$$
\begin{aligned}
L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2} \\
&= \Theta(n\lg n)
\end{aligned}
$$

4.4.4　程序实现

1. 复数类型及其运算

由于对多项式的 FFT 涉及复数计算,所以必须先定义复数数据类型并实现关于复数的算术运算。

```
typedef struct{
    double real;
    double magic;
}Complex;
Complex compSum(Complex a,Complex b){
    Complex c;
    c.real=a.real+b.real;
    c.magic=a.magic+b.magic;
    return c;
}
Complex compDiff(Complex a,Complex b){
```

```
12     Complex c;
13     c.real=a.real-b.real;
14     c.magic=a.magic-b.magic;
15     return c;
16 }
17 Complex compProd(Complex a,Complex b){
18     Complex c;
19     c.real=a.real*b.real-a.magic*b.magic;
20     c.magic=a.real*b.magic+a.magic*b.real;
21     return c;
22 }
23 Complex compQuot(Complex a,Complex b){
24     Complex c;
25     double r=b.real*b.real+b.magic*b.magic;
26     assert(r);
27     c.real=(a.real*b.real+a.magic*b.magic)/r;
28     c.magic=(a.magic*b.real-a.real*b.magic)/r;
29     return c;
30 }
```

程序 4-5　复数类型的定义及其算术运算函数

对程序 4-5 的说明如下。

(1) 第 1～4 行将复数定义成具有 2 个 **double** 型属性 real、magic 的结构体类型 Complex。其中属性 real、magic 分别表示复数的实部和虚部。

(2) 第 5～10 行及第 11～16 行定义的函数 compSum 及 compDiff 分别实现复数的加法与减法运算：$(a_1+b_1i)\pm(a_2+b_2i)=(a_1\pm a_2)(b_1\pm b_2)i$。

(3) 第 17～22 行定义的函数 compProd 实现复数的乘法运算 $(a_1+b_1i)\cdot(a_2+b_2i)=(a_1\cdot a_2-b_1\cdot b_2)+(a_1\cdot b_2+b_1\cdot a_2)i$。

(4) 第 23～30 行定义的函数 compQuot 实现复数的除法运算 $(a_1+b_1i)\div(a_2+b_2i)=((a_1\cdot a_2+b_1\cdot b_2)+(b_1\cdot a_2-a_1\cdot b_2))/(a_2{}^2+b_2{}^2)$。

程序 4-5 中，Complex 类型的定义及各函数的原型声明保存于文件夹 algebra 中的头文件 complex.h 中，各函数定义保存于同一文件夹的源文件 complex.c 中。

2. 计算 FFT

仅实现计算 4-12 中计算 FFT 算法的迭代版本。

```
static Complex* bitReverseCopy(Complex *a,int n){
    int bit,i,k,lgn,c,d;
    Complex *A=(Complex*)calloc(n,sizeof(Complex));
    lgn=lg(n);                              /*计算 n 的二进制位数*/
    for(k=0;k<n;k++){
        bit=1;c=0;
        for(i=0;i<lgn;i++)                  /*对 k 逐位转换到对称位得到 c*/
            if(k&bit){                      /*k 的第 i 位为 1*/
```

```
9                  d=1;c|=(d<<=(lgn-i-1));        /* 将 d 添加到 c 的对应位上 */
10             bit<<=1;                           /* bit 左移 1 位 */
11         }
12         A[c]=a[k];
13     }
14     return A;
15 }
16 Complex* fft(Complex *a,int n,int inv){         /* 非递归的 FFT */
17     int s,m=1,lgn,j,k;
18     Complex *A=bitReverseCopy(a,n);             /* A 赋值为 a 的比特反转置换 */
19     lgn=lg(n);                                  /* 计算 n 的二进制位数 */
20     for(s=1;s<=lgn;s++){
21         Complex wm;
22         m*=2;
23         wm.real=cos(2*PI/m);wm.magic=inv*sin(2*PI/m);
24         for(k=0;k<n;k+=m){
25             Complex w={1.0,0.0},t,u;
26             for(j=0;j<m/2;j++){
27                 t=compProd(w,A[k+j+m/2]);       /* t← ω A[k+j+m/2] */
28                 u=A[k+j];                       /* u←A[k+j] */
29                 A[k+j]=compSum(u,t);            /* A[k+j]←u+t */
30                 A[k+j+m/2]=compDiff(u,t);       /* A[k+j+m/2]←u-t */
31                 w=compProd(w,wm);               /* 计算旋转因子 */
32             }
33         }
34     }
35     return A;
36 }
```

程序 4-6　实现计算向量 a 的 FFT 算法的函数

对程序 4-6 的说明如下。

（1）第 1～15 行定义的函数 bitReverseCopy 实现算法 4-13，计算具有 n 个元素的数组 a 的比特反转，将计算结果作为函数值返回。其中，第 4 行调用函数 lg(n)计算 n 的二进制位数。该函数定义在文件夹 algebra 的源文件 fft.c 中，读者可打开文件研读。第 5～13 行的 **for** 循环实现算法 4-13 中的第 2～4 行的 **for** 循环。循环体中第 6～11 行的操作实现的是算法 4-13 中的操作 $c \leftarrow rev(k)$。基本思想是将 k 的从低到高的非 0 各位（bit 从 1 开始与 k 按位与 k&bit，每次重复的最后将 bit 左移 1 位 bit<<=1）置为 c 的从高到低的各对应位(d=1;c|=(d<<=(lgn-i-1)))。

（2）第 14～36 行定义的函数 fft 实现算法 4-12，快速计算 n 维向量 a 的 DFT。注意，int 型参数 inv 为 1 时，正常计算 a 的 DFT；inv 为 -1 时，该函数计算 DFT^{-1}。

（3）在 fft 的函数体中，第 18 行调用函数 bitReverseCopy，计算 a 的比特反转，结果保存于 Complex 型的数组 A 中，完成算法 4-12 中的第 1 行操作 BIT-REVERSE-COPY(a,A)。第 20～34 行的 **for** 循环实现算法 4-12 中第 3～11 行的 **for** 循环。其中第 23 行完成算法中

对旋转因子的初始化操作 $\omega_m \leftarrow e^{\pm 2\pi i/m}$,(计算 DFT^{-1}时指数应为负)式中的 m 值在每次重复时通过迭代 m * =2,置为 2^s。这是根据数学中复数的表达式 $e^{\pm 2\pi i/m}=\cos 2\pi/m \pm i\sin 2\pi/m$ 而得。内嵌于该循环中的第 24～33 行的 **for** 循环对应于算法 4-12 中的第 6～11 行的 **for** 循环。其中,第 25 行中将复单位根 w 初始化为{1.0,0.0},对应于算法中的 $\omega \leftarrow 1$ 操作。其余部分的操作与算法中的几乎一一对应。要注意的是,由于进行的算术运算是对复数进行的,所以加、减、乘应分别调用函数 compSum、compDiff、compProd 来完成。

程序 4-6 中的函数定义于文件夹 algebra 中的源文件 fft.c 中,fft 的原型声明保存在同一文件夹内的头文件 fft.h 中。

3. 计算 FFT^{-1}

尽管在函数 fft 中用参数 inv 来控制计算 DFT 还是 DFT^{-1},当 inv 为－1 时,调用 fft 还不能完成 DFT^{-1}的计算,因为还需要对得到的向量除以 n。这由下列的函数来完成。

```
1 Complex * fftInverse(Complex * a,int n){        /* 非递归 DFT 逆 */
2   Complex * b,x={1.0/n,0.0};
3   int i;
4   b=fft(a,n,-1);
5   for(i=0;i<n;i++)
6     b[i]=compProd(b[i],x);
7   return b;
8 }
```

程序 4-7 完成 DFT^{-1}计算的函数

该函数的第 2 行将 x 置为 1/n。在第 4 行计算出 b=fft(a,n,－1)后,第 5 行和第 6 行的 for 循环完成对向量 b 除以 n 的操作。

函数 fftInverse 也定义于源文件 fft.c 中,其原型声明也在头文件 fft.h 中。

4. 多项式及其算术运算

利用定义好的复数类型及对复数的算术运算函数及对向量的 FFT 变换函数,将多项式定义为如下的结构体类型。

```
typedef struct{                                  /* 多项式类型 */
    Complex * coeff;                             /* 系数表 */
    int degree;                                  /* 次数 */
}Polynomial;
double horner(Polynomial a,double x){           /* 霍纳法计算多项式的值 */
    Complex y={0.0,0.0},x0={x,0.0};
    int i,n=a.degree;
    for(i=n;i>=0;i--)
        y=compSum(a.coeff[i],compProd(x0,y));
    return y.real;
}
Polynomial polySum(Polynomial a,Polynomial b){  /* 多项式的和 */
```

```
13     int m=a.degree,n=b.degree,i=0,j=0,k=0;
14     Polynomial c=newPoly((m>n? m:n)+1);            /*和多项式*/
15     while(i<=m&&j<=n)
16         c.coeff[k++]=compSum(a.coeff[i++],b.coeff[j++]);
17     while(i<=m)
18         c.coeff[k++]=a.coeff[i++];
19     while(j<=n)
20         c.coeff[k++]=b.coeff[j++];
21     k--;
22     while(k>0&&isZero(c.coeff[k--]))               /*去掉高次项0系数*/
23         c.degree--;
24     return c;
25 }
26 static Complex* componentwiseMult (Complex* f,Complex* g,int n){      /*按分量积*/
27     Complex* p=(Complex*)calloc(n,sizeof(Complex));
28     int i;
29     for(i=0;i<n;i++)
30         p[i]=compProd(f[i],g[i]);
31     return p;
32 }
33 Polynomial polyProd(Polynomial f,Polynomial g){              /*多项式相乘*/
34     int L,n,max,min;
35     Polynomial p;
36     Complex *a,*b,*x,*y,*z;
37     max=f.degree>=g.degree? f.degree:g.degree;
38     min=f.degree<=g.degree? f.degree:g.degree;
39     L=max+min+1;
40     n=1;
41     while(n<1)n*=2;
42     assert(a=(Complex*)calloc(n,sizeof(Complex)));
43     memcpy(a,f.coeff,(f.degree+1)*sizeof(Complex));
44     assert(b=(Complex*)calloc(n,sizeof(Complex)));
45     memcpy(b,g.coeff,(g.degree+1)*sizeof(Complex));
46     x=fft(a,n,1);y=fft(b,n,1);
47     z=componentwiseMult (x,y,n);
48     p.coeff=fftInverse(z,n);
49     p.degree=min+max;
50     free(a);free(b);free(x);free(y);free(z);
51     return p;
52 }
```

程序 4-8　多项式类型定义及多项式的运算函数

对程序 4-8 的说明如下。

(1) 第 1～4 行将多项式定义成具有 2 个属性的结构体类型 Polynomial。其中，Complex 型指针 coeff 指引表示系数的数组，**int** 型 degree 表示多项式的次数。之所以要将

系数向量定义成 Complex 型的数组,是因为要使用 FFT 计算多项式的乘法。实系数多项式的每个系数可以用虚部为 0 的复数表示。

(2) 第 5~11 行定义的函数 horner 实现算法 4-8,用霍纳法计算多项式在指定子变量值 x 处的值。由于多项式的系数表示成了复数,所以第 6 行将计算结果 y 初始化为{0.0, 0.0},且用{x,0.0}初始化 x0,便于跟多项式的系数进行算术运算。程序代码与算法伪代码的结构是一致的,只是要调用复数的运算函数来计算相应的算术运算。

(3) 第 12~25 行定义的函数 polySum 实现算法 4-9,计算两个多项式 a、b 的和,并将其作为函数值返回。其中,第 15 行和第 16 行的 **while** 循环计算 a、b 对应项系数的和,作为和多项式对应项的系数。第 17 行和第 18 行处理 a 的次数高于 b 的次数时,次数高于 n 的剩余项系数。类似地,第 19 行和第 20 行处理 b 的次数高于 a 的次数的情形。由于两个同次多项式相加可能造成高次幂系数为 0,第 22 行和第 23 行处理去掉系数为 0 的高次项。

(4) 第 33~52 行定义的函数 polyProd,利用对向量的 FFT 变换计算多项式 f 和 g 的积,并将其作为函数值返回。其中,第 39 行计算积多项式的系数数组长度 L,第 41 行最接近于 L 的 2 的整数幂,记于 n。第 42~45 行将 f、g 的系数向量分别扩展成长度为 n 的数组 a、b。第 46 行调用函数 fft 分别计算向量 a 和 b 的 DFT 记为 x、y。第 47 行调用函数 componentwiseMult,计算 x、y 的点值积,记为 z,该函数定义于第 26~32 行。第 48 行调用函数 fftInverse 计算 z 的 DFT^{-1},作为多项式 p 的系数向量。第 49 行设置 p 的次数。至此,完成了定理 4-8 中计算多项式积

$$\boldsymbol{a} \otimes \boldsymbol{b} = \text{DFT}_{2n}{}^{-1}(\text{DFT}_{2n}(\boldsymbol{a}) \cdot \text{DFT}_{2n}(\boldsymbol{b}))$$

的全部工作。第 50 行释放各中间临时向量 a、b、x、y、z 的空间,51 行返回计算结果多项式 p。

程序 4-8 中定义的 Polynomial 类型及关于 Polynomial 型数据的各算术运算函数的原型声明存储于 algebra 文件夹中的头文件 polynomial.h 中,各函数的定义存储于同一文件夹中的源文件 polynomial.c 内。

顺便说明一下,程序 4-8 中的 Polynomial 类型为适应 FFT 变换中复数的运算,所以将系数向量定义成 Complex 型的数组。如果将多项式的乘法实现为算法 4-10 那样的向量的卷积,则可将系数向量定义成 **double** 型数组。笔者比照程序 4-8,实现了实系数多项式类型 Poly 以及对 Poly 型数据的算术运算函数。代码存储于 algebra 文件夹中的头文件 poly.h 和 poly.c 中,读者可打开文件参考。

4.5 应用

4.5.1 多项式的泰勒展开式

Equivalent Polynomial

Description

Given a polynomial $\sum_{k=0}^{n} a_k x^k, a_n \neq 0$ and a number t, please convert it into an equivalent polynomial in the form of $\sum_{k=0}^{n} b_k (x-t)^k, b_n \neq 0$.

Input

The input contains several test cases.

The first line of each test case gives two integer n ($1<n\leqslant 200$) and t ($-10\leqslant t\leqslant 10$). The following line is a sequence of n integer $a_0, a_1, \cdots, a_n$ ($-1000\leqslant a_i\leqslant 1000$), which is separated by exactly one space.

Output

For each test case, output the equivalent polynomial's coefficient $b_0, b_1, \cdots, b_n$. One line for each test case and each number is separated by exactly one space, no extra space at the end of each line.

Sample Input

```
2 1
1 0 2
```

Sample Output

```
3 4 2
```

1. 问题的理解与分析

对给定的多项式函数 $f(x)=\sum_{k=0}^{n}a_kx^k, a_n\neq 0$ 及常数 t，假定有另一个 n 次多项式 $g(x)=\sum_{k=0}^{n}b_k(x-t)^k, b_n\neq 0$ 与之相等，即 $\sum_{k=0}^{n}a_kx^k=\sum_{k=0}^{n}b_k(x-t)^k$。要求计算出 $g(x)$，即计算出 $g(x)$ 的系数向量 $(b_0, b_1, \cdots, b_n)$。

将多项式 $f(x)$ 看成 x 的函数，数学分析告诉我们 $f(x)$ 对 x 的各阶导数有着非常“优良”的结构：

$$
\begin{aligned}
&f^{(0)}(x)=f(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n\\
&f'(x)=a_1+2a_2x+3a_3x^2+\cdots+na_nx^{n-1}\\
&f^{(2)}(x)=(f'(x))'=2!a_2+3\cdot 2a_3x+\cdots+n(n-1)a_nx^{n-2}\\
&\vdots\\
&f^{(k)}(x)=(f^{(k-1)}(x))'=k!a_2+\cdots+n(n-1)\cdots(n-k+1)a_nx^{n-k}\\
&\vdots\\
&f^{(n)}(x)=(f^{(n-1)}(x))'=n!a^n\\
&f^{(n+1)}(x)=0\\
&\vdots
\end{aligned}
$$

可以用下列过程对由 $f=(a_0, a_1, \cdots, a_n)$ 给定的所项式，计算它的导数。

```
DERIVATIVE(f)
1 for i ← 1 to n
2    do b_{i-1} ← i · a_i
3 return f' = (b_0, b_1, …, b_{n-1})
```

算法 4-14　计算多项式 $f(x)$ 导数的过程

回到问题本身。由 $f(x)=\sum_{k=0}^{n}a_kx^k=\sum_{k=0}^{n}b_k(x-t)^k=g(x)$，有

$$\left.\begin{array}{l} f(t)=b_0 \\ f'(t)=b_1 \\ f^{(2)}(t)=2!b_2 \\ \vdots \\ f^{(k)}(t)=k!b_k \\ \vdots \\ f^{(n)}(t)=n!b_n \end{array}\right\} \Rightarrow \left\{\begin{array}{l} b_0=f(t) \\ b_1=f'(t) \\ b_2=f^{(2)}(t)/2! \\ \vdots \\ b_k=f^{(k)}(t)/k! \\ \vdots \\ b_n=f^{(n)}(t)/n! \end{array}\right.$$

即 $f(x)=g(x)=\sum_{k=0}^{n}b_k(x-t)^k=\sum_{k=0}^{n}\frac{f^{(k)}(t)}{k!}(x-t)^k$。这恰是 $f(x)$在 $x=t$ 处的泰勒展开式。也就是说，对给定的多项式 $f(x)=\sum_{k=0}^{n}a_kx^k$ 和数值 t，计算出 $f(x)$在 $x=t$ 处的泰勒展开式，就解决本问题的要求。利用 DERIVATIVE 过程，可以写出如下的计算 $f(x)$在$x=t$ 处的泰勒展开式的算法。

```
TYLER(f,t)
factor ← 1, f1 ← f                  ▷k 的阶乘 factor 初始化为 1，k 阶导数 f1 初始化为 f
for k ← 0 to n
    do if k>0
          then factor ← factor · k
       b_k ← HORNER(f1,t)/factor     ▷b_k = f^(k)(t)/k!
       f1 ← DERIVATIVE(f1)           ▷计算 k+1 阶导函数
return g=(b0,b1,…,bn)
```

算法 4-15 多项式 $f(x)$在 $x=t$ 处的泰勒展开式的过程

2. 程序实现

```
Poly derivative(Poly f){
    int n=f.degree,k;
    Poly f1=newPoly(n);
    for(k=1;k<=n;k++)
        f1.coeff[k-1]=f.coeff[k] * k;
    return f1;
}
Poly tyler(Poly f,double t){
    int k,n=f.degree;
    Poly f1=newPolyByArry(f.coeff,n+1),tf={NULL,-1},g=newPoly(n+1);
    double factor=1.0;
    for(k=0;k<=n;k++){
        if(k!=0)
            factor *=k;
        g.coeff[k]=horner(f1,t)/factor;
        polyAssign(&tf,f1);clrPoly(f1);
```

```
17          f1=derivative(tf);
18      }
19      clrPoly(f1);clrPoly(tf);
20      return g;
21 }
22 int main(void){
23      double t;
24      Poly g={NULL,-1} f={NULL,-1};
25      int k,n;
26      FILE *f1=fopen("chap04/Equivalent Polynomial/inputdata.txt","r"),
27           *f2=fopen("chap04/Equivalent Polynomial/outputdata.txt","w");
28      assert(f1&&f2);
29      while(!feof(f1)){
30          fscanf(f1,"%d%lf",&n,&t);
31          if(f.coeff)clrPoly(f);
32          f=newPoly(n+1);
33          for(k=0;k<=n;k++)
34              fscanf(f1,"%lf",f.coeff+k);
35          if(g.coeff)clrPoly(g);
36          g=tyler(f,t);
37          for(k=0;k<=n;k++)
38              fprintf(f2,"%.0f ",b.coeff[k]);
39          fputc('\n',f2);
40      }
41      clrPoly(f);clrPoly(g);
42      fclose(f1);fclose(f2);
43      return 0;
44 }
```

程序 4-9　解决 Equivalent Polynomial 问题的 C 程序

对程序 4-9 的说明如下。

(1) 第 1～7 行定义的函数 derivative 实现算法 4-14，计算多项式函数 $f(x)$的导函数 $f'(x)$。程序代码结构与算法伪代码结构很接近。需要注意的是，该函数的形参 f 是 Poly 型数据，且函数返回值也是 Poly 型的。Poly 类型定义在文件夹 algebra 中的 poly.h 中。

(2) 第 8～21 行定义的函数 tyler，实现算法 4-15，计算多项式函数 $f(x)$在 $x=t$ 处的泰勒展开式。程序代码与算法伪代码结构十分接近。需要注意的是，Poly 型数据的 coeff 属性是由指针指引的动态内存。所以，在第 17 行对 Poly 型变量 f1 重新赋值之前，需要在第 16 行中调用 clrPoly 函数，释放已占有空间，以防内存泄漏。

(3) 第 22～44 行定义的 main 函数利用 Poly 类型并调用 tyler 函数，解决 Equivalent Polynomial 问题。第 29～40 行的 **while** 循环对存储于由文件指针 f1 指引的输入文件中的每个案例数据重复计算多项式在指定点处的泰勒展开式。其中，第 30 行从 f1 中读取多项式 f 的次数 n 和自变量的指定值 t。第 31～35 行在 f1 中读取多项式 f 的系数向量。其中第 31 行和第 32 行对 f 释放已占有的空间，理由与(2)中对 f1 重新赋之前释放空间的一样，请

参阅。第 35 行和第 36 行调用函数 tyler 计算 f 在 t 处的泰勒展开式 g。注意第 35 行对 g 所做的释放空间的操作。第 37～39 行输出 g。

程序 4-9 存储于文件夹 chap04\Equivalent Polynomial 中的源文件 EquivalentPolynomial.c。

4.5.2 完善序列

Complete the sequence!

You probably know those quizzes in Sunday magazines: given the sequence 1,2,3,4,5,what is the next number? Sometimes it is very easy to answer,sometimes it could be pretty hard. Because these " sequence problems" are very popular, ACM wants to implement them into the "Free Time" section of their new WAP portal.

ACM programmers have noticed that some of the quizzes can be solved by describing the sequence by polynomials. For example, the sequence 1, 2, 3, 4, 5 can be easily understood as a trivial polynomial. The next number is 6. But even more complex sequences,like 1,2,4,7,11,can be described by a polynomial. In this case,$1/2 \cdot n^2 - 1/2 \cdot n + 1$ can be used. Note that even if the members of the sequence are integers,polynomial coefficients may be any real numbers.

Polynomial is an expression in the following form:

$$P(n) = a_D \cdot n^D + a_{D-1} \cdot n^{D-1} + \cdots + a_1 \cdot n + a_0$$

If $a_D \neq 0$,the number D is called a degree of the polynomial. Note that constant function $P(n)=C$ can be considered as polynomial of degree 0,and the zero function $P(n)=0$ is usually defined to have degree -1.

Input

There is a single positive integer T on the first line of input. It stands for the number of test cases to follow. Each test case consists of two lines. First line of each test case contains two integer numbers S and C separated by a single space,$1 \leqslant S < 100$,$1 \leqslant C < 100$,$(S+C) \leqslant 100$. The first number,S,stands for the length of the given sequence,the second number,C is the amount of numbers you are to find to complete the sequence.

The second line of each test case contains S integer numbers $X_1, X_2, \cdots, X_S$ separated by a space. These numbers form the given sequence. The sequence can always be described by a polynomial $P(n)$ such that for every i, $X_i = P(i)$. Among these polynomials, we can find the polynomial $P_{\min}$ with the lowest possible degree. This polynomial should be used for completing the sequence.

Output

For every test case, your program must print a single line containing C integer numbers,separated by a space. These numbers are the values completing the sequence according to the polynomial of the lowest possible degree. In other words,you are to print values $P_{\min}(S+1), P_{\min}(S+2), \cdots, P_{\min}(S+C)$.

It is guaranteed that the results $P_{\min}(S+i)$ will be non-negative and will fit into the standard integer type.

Sample Input

```
4
6 3
1 2 3 4 5 6
8 2
1 2 4 7 11 16 22 29
10 2
1 1 1 1 1 1 1 1 1 2
1 10
3
```

Sample Output

```
7 8 9
37 46
11 56
3 3 3 3 3 3 3 3 3 3
```

1. 问题的理解与分析

对于一合法案例数据 $C,S,y_1,y_2,\cdots,y_C$，要求计算出一个多项式 $p(x)$，使得该多项式满足 $p(i)=y_i, i=1,2,\cdots,S$，并用此多项式计算输出 $p(S+1),p(S+2),\cdots,p(S+C)$。

设 $p(x)=p_0+p_1x+\cdots+p_{C-1}x^{C-1}$，则 $p(i)=b_i, i=1,2,\cdots,S$，对应一个以 $p_0,p_1,\cdots,p_{S-1}$ 为未知数的方程组：

$$\begin{cases} p_0+p_1\cdot 1+p_2 1^2+\cdots p_{S-1}1^{S-1} & =b_1 \\ p_0+p_1\cdot 2+p_2 2^2+\cdots p_{S-1}2^{S-1} & =b_2 \\ \quad\quad\quad\vdots & \quad\vdots \\ p_0+p_1\cdot S+p_2 S^2+\cdots p_{S-1}S^{S-1} & =b_S \end{cases}$$

该方程组的系数矩阵是一个 Vandermonde 矩阵：

$$\mathbf{V}=\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2^{S-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & S & \cdots & S^{S-1} \end{pmatrix}$$

对已知正整数 S，可以用以下过程创建这一矩阵。

```
VANDERMONDE(S)
for i ← 1 to S
    do V[i,1]← 1
       t ← 1
       for j ← 2 to S
           do t ← t · j
```

```
            V[i,j]← t
return V
```

算法 4-16 生成 S 阶 Vandermonde 矩阵的过程

Vandermonde 矩阵是一个可逆阵,该方程组有唯一解 $\boldsymbol{p}=(a_0,a_1,\cdots,a_{S-1})^T$,利用 LUP-DECOMPOSITION 对 **V** 进行 LUP 分解,并调用 LUP-SOLVE 解方程组 $\boldsymbol{Vp}=\boldsymbol{b}$,其中 $\boldsymbol{y}=(b_1,b_2,\cdots,b_S)^T$,就能确定多项式 $p(x)=a_0+a_1x+\cdots+a_{S-1}x^{S-1}$。得到了 $p(x)$后,调用过程 HORNER,就可计算并输出 $p(S+j)$,$j=1,2,\cdots,S$,问题从而得以解决。

2. 程序实现

```
Matrix vandermonde(int s){
    Matrix v=newMatrix(s,s);
    int i,j;
    double t;
    for(i=1;i<=s;i++){
        v.tab[(i-1)*s]=t=1.0;
        for(j=2;j<=s;j++)
            v.tab[(i-1)*s+j-1]=t=t*i;
    }
    return v;
}
int main(){
    double *b=NULL;
    int n,t,s,c,*pi=NULL,i,k;
    Poly p={NULL,-1};
    Matrix a={NULL,0,0},lu={NULL,0,0};
    FILE *f1=fopen("chap04/Complete the sequence/inputdata.txt","r"),
        *f2=fopen("chap04/Complete the sequence/outputdata.txt","w");
    assert(f1&&f2);
    fscanf(f1,"%d",&t);                          /*读取案例数*/
    for(k=0;k<t;k++){                            /*对每一个案例*/
        fscanf(f1,"%d%d",&s,&c);                 /*读取案例中的数据 s、c*/
        if(b)free(b);
        if(pi)free(pi);
        assert(b=(double*)calloc(s,sizeof(double)));
        assert(pi=(int*)calloc(s,sizeof(int)));
        for(i=0;i<s;i++)                         /*读取向量 b*/
            fscanf(f1,"%lf",b+i);
        if(a.tab)clrMatrix(a);
        a=vandermonde(s);                        /*构造 Vandermonde 矩阵*/
        if(lu.tab)clrMatrix(lu);
        lu=lupDecomposition(a,pi);               /*对 a 做 LUP 分解*/
        if(p.coeff)free(p.coeff);
        p.degree=s-1;
```

```
35          p.coeff=lupSolve(lu,pi,b);                  /* 解方程组 ap=b */
36          nomalPoly(&p);
37          for(i=0;i<c;i++)                            /* 计算并输出 p(s+1),…,p(s+c) */
38              fprintf(f2,"%.0f ",horner(p,i+s+1.0));
39          fputc('\n',f2);
40      }
41      free(b);free(pi);
42      clrMatrix(a);clrMatrix(lu);
43      clrPoly(p);
44      fclose(f1);fclose(f2);
45      return 0;
45 }
```

程序 4-10　解决 Complete the sequence 问题的 C 程序

对程序 4-10 的说明如下。

(1) 第 1～11 行定义的函数 vandermonde,实现算法 4-16 计算 s 阶 vandermonde 矩阵,并作为函数值返回。注意,利用程序 4-1 中定义的类型 Matrix 表示矩阵。

(2) 第 12～45 行定义的 main 函数将调用 vandermonde 函数,程序 4-3 中定义的 lupDecomposition 函数,程序 4-4 中定义的 lupSolve 函数解决 Complete the sequence 问题。问题的输入数据包含于文件指针 f1 指向的磁盘文件 inputdata.txt 中,输出数据记录在文件指针 f2 指向的磁盘文件 outputdata.txt 中。

(3) 在 main 函数中,第 20 行读取输入文件 f1 中的案例数 t。第 21～40 行的 **for** 循环重复处理 t 个案例。其中,第 22 行在 f1 中读取案例数据 s 和 c。第 27 行和第 28 行在 f1 中读取向量 b。第 30 行调用函数 vandermonde 构造 s 阶 vandermonde 矩阵 a。第 32 行调用 lupDecomposition 函数对 a 进行 LUP 分解,结果存于矩阵 lu 及向量 pi 中。第 35 行调用函数 lupSolve,利用 lu 及 pi 解方程组 ap=b。第 37 行和 38 行的 **for** 循环调用函数 horner,计算输出 p(s+1),…,p(s+c)。

程序 4-10 存储在文件夹 ch04\Complete the sequence 的源文件 Complete the sequence.c 中。

4.5.3　函数的有理式逼近

Rational Approximation

A polynomial $p(x)$ of degree n can be used to approximate a function $f(x)$ by setting the coefficients of $p(x)$ to match the first n coefficients of the power series of $f(x)$ (expanded about $x=0$). For example,

$$1/(1-x) \approx 1 + x + x^2 + \cdots + x^n$$

Unfortunately, polynomials are "nice" and they do not work well when they are used to approximate functions that behave poorly (e.g. those with singularities). To overcome this problem, we can instead approximate functions by rational functions of the form $p(x)/q(x)$, where $p(x)$ and $q(x)$ are polynomials. You have been asked by Approximate

Calculation Machinery to solve this problem, so they can incorporate your solution into their approximate calculation software.

Given m, n, and the first $m+n$ coefficients of the power series of $f(x)$, we wish to compute two polynomials $p(x)$ and $q(x)$ of degrees at most $m-1$ and $n-1$, respectively, such that the power series expansion of $q(x) \cdot f(x) - p(x)$ has 0 as its first $m+n-1$ coefficients, and 1 as its coefficient corresponding to the x^{m+n-1} term. In other words, we want to find $p(x)$ and $q(x)$ such that

$$q(x) \cdot f(x) - p(x) = x^{m+n-1} + \cdots$$

where ... contains terms with powers of x higher than $m+n-1$. From this, $f(x)$ can be approximated by $p(x)/q(x)$.

Background Definitions

A polynomial $p(x)$ of degree n can be written as $p_0 + p_1x + p_2x^2 + \cdots + p_nx^n$, where p_i's are integers in this problem.

A power series expansion of $f(x)$ about 0 can be written as $f_0 + f_1x + f_2x^2 + \cdots$, where f_i's are integers in this problem.

Input

The input will consist of multiple cases. Each case will be specified on one line, in the form m n f_0 $f_1 \cdots$ f_{m+n-1} where f_i is the coefficient of x^i in the power series expansion of f. You may assume that $1 \leqslant m$, $1 \leqslant n \leqslant 4$, $2 \leqslant m+n \leqslant 10$, and f_i are integers such that $|f_i| \leqslant 5$. The end of input will be indicated by a line containing $m=n=0$, and no coefficients for f. You may assume that there is a unique solution for the given input.

Output

For each test case, print two lines of output. Print the polynomial $p(x)$ on the first line, and then $q(x)$ on the second line. The polynomial $p(x)$ should be printed as a list of pairs (p_i, i) arranged in ascending order in i, such that p_i is a non-zero coefficient for the term x^i. Each non-zero coefficient p_i should be printed as a/b, where $b>0$ and a/b is the coefficient expressed in lowest terms. In addition, if $b=1$ then print only a (and omit b). If $p(x)=0$, print a line containing only (0,0). Separate the pairs in the list by one space. The polynomial $q(x)$ should be printed in the same manner. Insert a blank line between cases.

Sample Input

```
2 2 0 0 1 1
4 2 1 2 3 4 5 -2
1 1 2 3
1 4 -5 0 -2 1 -2
0 0
```

Sample Output

```
(0,0)
```

```
(1,1)

(-4/33,0) (-1/11,1) (-2/33,2) (-1/33,3)
(-4/33,0) (5/33,1)

(2/3,0)
(1/3,0)

(25/6,0)
(-5/6,0) (1/3,2) (-1/6,3)
```

1. 问题的理解与分析

已知函数 $f(x)=f_0+f_1x+\cdots+f_{m+n-1}x^{m+n-1}$，计算多项式 $p(x)=p_0+p_1x+\cdots+p_{m-1}x^{m-1}$ 及 $q(x)=q_0+q_1x+\cdots+q_{n-1}x^{n-1}$，使得

$$q(x)\cdot f(x)-p(x)=x^{m+n-1}+\cdots$$

若能算出 $p(x)$ 与 $q(x)$，在数学上认为 $f(x)$ 在 $x=0$ 的附近可以由多项式 $p(x)$ 与 $q(x)$ 的商 $p(x)/q(x)$ 逼近。由于

$$\begin{aligned}q(x)\cdot f(x)-p(x)&=\left(\sum_{i=0}^{n-1}q_ix^i\right)\left(\sum_{i=0}^{m+n-1}f_ix^i\right)-\sum_{i=0}^{m-1}p_ix^i\\&=\sum_{i=0}^{m+n-1}\sum_{j=0}^{i}q_jf_{i-j}x^i-\sum_{i=0}^{m-1}p_ix^i\qquad(\text{利用式(4-20)，对 } j\geqslant n,q_j=0)\\&=\sum_{i=0}^{m+n-1}\left(\sum_{j=0}^{i}q_jf_{i-j}-p_i\right)x^i\qquad(\text{对 } i\geqslant m,p_i=0)\end{aligned}$$

要使 $q(x)\cdot f(x)-p(x)=x^{m+n-1}+\cdots$ 成立，需

$$\begin{cases}\displaystyle\sum_{j=0}^{i}q_jf_{i-j}-p_i=0 & i=0,1,\cdots,m+n-2\\ \displaystyle\sum_{j=0}^{m+n-1}q_jf_{i-j}-p_{m+n-1}=1\end{cases}$$

这是一个关于 $m+n$ 个未知数 $q_0,\cdots,q_{n-1},p_0,\cdots,p_{m-1}$ 的线性方程组。该方程组的系数矩阵 $\boldsymbol{M}_{(m+n)\times(m+n)}$ 形如

$$\boldsymbol{M}=\begin{pmatrix}f_0&0&0&\cdots&0&-1&0&0&\cdots&0\\f_1&f_0&0&\cdots&0&0&-1&0&\cdots&0\\f_2&f_1&f_0&\cdots&0&0&0&-1&\cdots&0\\\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots\\f_{m-1}&f_{m-2}&\cdots&f_0&0&0&\cdots&\cdots&\cdots&-1\\f_m&f_{m-1}&\cdots&\cdots&\cdots&0&\cdots&\cdots&\cdots&0\\\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots\\f_{m+n-3}&f_{m+n-4}&\cdots&\cdots&f_{m-2}&0&\cdots&\cdots&\cdots&0\\f_{m+n-2}&f_{m+n-3}&\cdots&\cdots&f_{m-1}&0&\cdots&\cdots&\cdots&0\\f_{m+n-1}&f_{m+n-2}&\cdots&\cdots&f_m&0&\cdots&\cdots&\cdots&0\end{pmatrix}$$

对给定的案例数据 $f=(f_0,f_1,\cdots,f_{m+n-1})$,$m$ 和 n,可用如下过程生成矩阵 $\boldsymbol{M}$。

```
FORM-MATRIX(m,n,f)
for i ← 0 to m+n-1
  do if i>n-1 then t ← n-1
     else t ← i
     for j ← 0 to t
        do M[i,j]← f_{i-j}
     if i<m then M[i,n+i]←-1
return M
```

算法 4-17　构成合法案例 m、n、f 的系数矩阵的过程

令 $\boldsymbol{x}=(q_0,\cdots,q_{n-1},p_0,\cdots,p_{m-1})^T$,$\boldsymbol{b}=(\underbrace{0,\cdots,0}_{m+n-1},1)^T$,调用过程 LUP-DECOMPOSITION 和 LUP-SOLVE 解线性方程组 $\boldsymbol{Mx}=\boldsymbol{b}$,即可得到案例的解 $\boldsymbol{p}=(p_0,\cdots,p_{m-1})$,$\boldsymbol{q}=(q_0,\cdots,q_{n-1})$。

2. 程序实现

1) 数据类型的定义

由于题目要求 $p_i(i=0,\cdots,m-1)$、$q_j(j=0,\cdots,n-1)$表示成有理数形式,所以首先需要定义有理数类型及对有理数算术运算。

```
1  typedef struct{
2      unsigned numerator;                            /* 分子 */
3      unsigned denominator;                          /* 分母 */
4      char sign;                                     /* 符号 0 表示正,1 表示负 */
5  }Rational;                                         /* 有理数类型 */
6  void printRational(Rational);                      /* 输出有理数 */
7  int rationalIsZero(Rational);                      /* 判断有理数是否为 0 */
8  int valueCompare(Rational a,Rational b);           /* 比较有理数绝对值的大小 */
9  Rational rationalSum(Rational,Rational);           /* 计算有理数的和 */
10 Rational rationalDiff(Rational,Rational);          /* 计算有理数的差 */
11 Rational rationalProd(Rational,Rational);          /* 计算有理数的积 */
12 Rational rationalQuot(Rational,Rational);          /* 计算有理数的商 */
```

程序 4-11　有理数类型的定义及算术运算函数声明

对程序 4-11 的说明如下。

(1) 有理数类型 Rational 定义成具有 3 个属性(分子、分母和符号)的结构体(第 1～5 行)。

(2) 第 6～8 行声明了 3 个有理数常规维护函数:打印输出的 printRational、检测 0 的 rationalIsZero 和比较两个有理数绝对值大小的 valueCompare。

(3) 第 9～12 行分别声明了对有理数进行加、减、乘、除运算的函数 rationalSum、rationalDiff、rationalProd 和 rationalQuot。

程序 4-11 的代码存储为 algebra 文件夹中的头文件 rational.h,各函数的定义代码存储

为同一文件夹中的源文件 rational. c。为节省篇幅，此处不再展开这些代码，读者可打开文件研读。

利用 Rational 类型，定义元素为 Rational 的矩阵类型。

```
1 typedef struct{
2     Rational *tab;                          /* 二维数表 */
3     int row,col;                            /* 行数,列数 */
4 }rMatrix;
5 rMatrix rnewMatrix(int,int);                /* 创建 0 矩阵 */
6 void rclrMatrix(rMatrix);                   /* 清理矩阵存储空间 */
7 void rswapRows(rMatrix a,int i,int j);      /* 交换两行 */
```

程序 4-12　有理数矩阵类型定义

若将程序 4-12 与程序 4-1 比较，读者会发现，Matrix 类型与 rMatrix 类型区别仅在于数表 tab 的元素类型不同而已。所以，对 rMatrix 类型数据的操作（第 5 行声明的矩阵创建操作 rnewMatrix、第 6 行声明的矩阵空间清理操作 rclrMatrix 及第 7 行声明的交换矩阵两行元素的操作 rswapRows）与对 Matrix 类型数据的相应操作十分相似，程序 4-12 的代码添加在 algebra 文件夹中的头文件 matrix. h 内，各函数的定义代码添加在同一文件夹中的源文件 matrix. c 中。

2）LUP 分解函数的改造

为解有理数线性方程组，我们要改造程序 4-3 和程序 4-4 中定义的函数 lupDecomposition 及 lupSolve，改造后的函数原型声明如下。

```
rMatrix rlupDecomposition(rMatrix a,int *pi);
Rational* rlupSolve(rMatrix lu,int *pi,Rational *b);
```

rlupDecomposition 及 rlupSolve 的定义代码与 lupDecomposition 及 lupSolve 的定义十分接近，仅将对浮点数的算术运算替换成程序 4-11 中声明的对有理数算术运算函数的调用，将交换 Matrix 两行的交换操作替换成对 rMatrix 两行的交换操作就可以了。为节省篇幅起见，此处不再赘述。上述这两个函数的原型声明添加在 algebra 文件夹中的头文件 lup. h 内，函数定义的代码添加在同一文件夹的源文件 lup. c 中。

3）问题的解

```
rMatrix formMatrix(int m,int n,int *f){
    int i,j,t;
    rMatrix M=rnewMatrix(m+n,m+n);
    for(i=0;i<m+n;i++){
        t=(i<n?i+1:n);
        for(j=0;j<t;j++){
            M.tab[i*(m+n)+j].numerator=(f[i-j]>=0?f[i-j]:-f[i-j]);
            if(f[i-j]<0)
                M.tab[i*(m+n)+j].sign=1;
        }
        if(i<m)
```

```
                M.tab[i*(m+n)+n+i].numerator=1,
                M.tab[i*(m+n)+n+i].sign=1;
        }
        return M;
}
int main(){
        int m,n;
        FILE *f1=fopen("chap04/Rational Approximation/inputdata.txt","r"),
             *f2=fopen("chap04/Rational Approximation/outputdata.txt","w");
        assert(f1&&f2);
        fscanf(f1,"%d%d",&m,&n);
        while(m!=0&&n!=0){                              /* 合法案例 */
            int i,*f,*pi,pm,qn;
            rMatrix a,lu;
            Rational *b,*x;
            assert(f=(int*)calloc(m+n,sizeof(int)));
            assert(b=(Rational*)calloc(m+n,sizeof(Rational)));
            assert(pi=(int*)calloc(m+n,sizeof(int)));
            for(i=0;i<m+n;i++)                          /* 设置向量 b=(0,…,0,1) */
                b[i].denominator=1;
            b[m+n-1].numerator=1;
            for(i=0;i<m+n;i++){                         /* 读取案例数据 f0,…,fm+n-1 */
                fscanf(f1,"%d",f+i);
                pi[i]=i;
            }
            a=formMatrix(m,n,f);free(f);                /* 构造系数矩阵 */
            lu=rlupDecomposition(a,pi);rclrMatrix(a);   /* 对 a 进行 LUP 分解 */
            x=rlupSolve(lu,pi,b);rclrMatrix(lu);        /* 解方程组 ax=b */
            pm=m+n-1;
            while(pm>n&&rationalIsZero(x[pm]))          /* 计算 p(x)的次数 */
                pm--;
            if(pm==n||!rationalIsZero(x[n])){           /* 输出 p(x) */
                fprintf(f2,"(");
                fprintRational(f2,x[n]);
                fprintf(f2,",%d) ",0);
            }
            for(i=n+1;i<m+n;i++)
                if(!rationalIsZero(x[i])){
                    fprintf(f2,"(");
                    fprintRational(f2,x[i]);
                    printf(f2,",%d) ",i-n);
                }
            fputc('\n',f2);
            qn=n-1;
            while(qn>0&&rationalIsZero(x[qn]))          /* 计算 q(x)的次数 */
```

```
57             qn--;
58         if(qn==0||!rationalIsZero(x[0])){          /* 输出 q(x) */
59             fprintf(f2,"(");
60             fprintRational(f2,x[0]);
61             fprintf(f2,",%d) ",0);
62         }
63         for(i=1;i<n;i++){
64             if(!rationalIsZero(x[i])){
65                 fprintf(f2,"(");
66                 fprintRational(f2,x[i]);
67                 fprintf(f2,",%d) ",i);
68             }
69         }
70         fprintf(f2,"\n\n");
71         free(x);free(pi);free(b);
72         fscanf(f1,"%d%d",&m,&n);                   /* 读取下一个案例 */
73     }
74     fclose(f1);fclose(f2);
75     return 0;
76 }
```

程序 4-13　解决 Rational Approximation 问题的 C 程序

对程序 4-13 的说明如下。

(1) 第 1～16 行定义的函数 formMatrix 实现算法 4-15，根据参数表示的案例数据 m、n、f 计算该案例对应方程组的系数矩阵。程序代码结构与算法的伪代码结构十分接近，读者可对照研读。

(2) 第 17～76 行定义的 main 函数将调用 formMatrix、rlupDecomposition、rlupSolve 等函数解决 Rational Approximation 问题。问题的输入数据包含于文件指针 f1 指向的磁盘文件 inputdata.txt 中，输出数据记录在文件指针 f2 指向的磁盘文件 f2 中。

(3) 在 main 函数中，第 23～73 行的 while 循环对每一个合法的案例数据(m，n，f)重复解得 p(x)和 q(x)。其中，第 30～32 行完成设置向量 $b=(0,\cdots,0,1)^T$ 的操作。第 33～36 行读取向量 $f=(f_0,f_1,\cdots,f_{m+n-1})$，并初始化表示置换矩阵的数组 pi。第 37 行调用函数 formMatrix，根据 m、n、f 构造线性方程组的系数矩阵 a，第 38 行调用 rlupDecomposition 对 a 进行 LUP 分解，得到矩阵 lu 和 pi。第 39 行调用函数 rlupSolve 解方程组 ax=b。得到的解 $x=(q_0,\cdots,q_{n-1},p_0,\cdots,p_{m-1})^T$。第 40～42 行计算 p(x)的次数 pm，根据 pm 的值决定是否输出 p_0(第 43～47 行)，第 48～53 行依次输出 $p_1,\cdots,p_{m-1}$ 中的非 0 项。第 55～69 行输出 q(x)，过程与上述输出 p(x)的类似，读者可对照研读。循环体中最后(第 72 行)在输入文件中读取下一个案例的数据 m、n。

程序 4-15 的代码存储于文件夹 chap04\Rational Approximation 的源文件 RationalApproximation.c 中。

第5章 计算几何

计算几何是计算机科学中研究解决几何问题算法的一个分支。在现代工程与数学中,计算几何应用于计算机图形学、机器人技术、大规模集成电路设计、计算机辅助设计以及统计学等多种领域。计算几何问题的输入往往是一个描述性的几何对象的集合,诸如点的集合、直线段的集合或是按时针方向排列的多边形顶点。输出通常是对那些对象询问的回应,如各条直线是否相交?或是另一个新的几何对象,如点集的凸壳(最小的封闭凸多边形)。

本章讨论一些平面上的计算几何算法。每一个输入的对象表示为点的集合$\{p_1,p_2,p_3,\cdots\}$,其中每一个$p_i=(x_i,y_i)$且$x_i,y_i\in\mathbf{R}$。例如,n个顶点的多边形P表示为其边界上的n个顶点按出现的顺序构成序列$\{p_0,p_1,p_2,\cdots,p_{n-1}\}$。计算几何也可以用于三维,甚至更高维的空间,然而这样的高维空间中的问题及其解都很难形象化,而在二维空间中却可以看到一个很好的例子。

5.1 线段的性质

本章的几个计算几何算法都要求回答关于线段的性质问题。两个不同的点$p_1=(x_1,y_1)$和$p_2=(x_2,y_2)$,任何满足,$0\leqslant\alpha\leqslant 1$,$x_3=\alpha x_1+(1-\alpha)x_2$,$y_3=\alpha y_1+(1-\alpha)y_2$的点$p_3=(x_3,y_3)$,称为$p_1$、$p_2$的**凸组合**。也可以写成$p_3=\alpha p_1+(1-\alpha)p_2$。直观地说,$p_3$是$p_1$和$p_2$连线上介于$p_1$和$p_2$之间的点。给定两个点$p_1$、$p_2$,**线段**$\overline{p_1p_2}$是$p_1$、$p_2$凸组合点的集合。称$p_1$,$p_2$为线段$\overline{p_1p_2}$的**端点**。有时,要考虑$p_1$、$p_2$的顺序,人们就称其为**有向线段**$\overrightarrow{p_1p_2}$。若$p_1$是原点$(0,0)$,人们把有向线段$\overrightarrow{p_1p_2}$视为向量$\boldsymbol{p}_2$。

在本节中,将探索以下问题。

(1) 给定两条有向线段$\overrightarrow{p_0p_1}$和$\overrightarrow{p_0p_2}$,$\overrightarrow{p_0p_1}$是否绕它们的公共端点p_0从$\overrightarrow{p_0p_2}$顺时针方向旋转而得?

(2) 给定线段$\overline{p_1p_2}$和$\overline{p_2p_3}$,如果先沿$\overline{p_1p_2}$行进再沿$\overline{p_2p_3}$行进是否需要在点p_2处左转弯?

(3) 线段$\overline{p_1p_2}$和$\overline{p_3p_4}$是否相交?

由于每个问题的输入规模都是$O(1)$,无疑均可在$O(1)$的时间内得到回答。此外,这个方法将仅使用加法、减法、乘法和比较运算。既不需要除法也不需要三角函数,这两种运算都很费时且容易产生舍入误差。例如,用“直接”方法判断两条线段是否相交——计算每条线段的直线方程$y=mx+b$(m为斜率,b为y轴上的截距),求出两直线的交点并检测该交点是否在两条线段中。这就需要使用除法,当两条线段几乎平行时,在实际的计算机中此问题对除法运算的精度是很敏感的。本节中的方法避免了除法,因此更加精确。

5.1.1　叉积及其应用

1. 叉积

叉积的计算是解决上述线段问题的方法之核心。考虑两个向量 $\boldsymbol{p}_1$ 和 $\boldsymbol{p}_2$，如图 5-1(a)所示。**叉积** $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 可以解释为由点 (0,0)、$\boldsymbol{p}_1$、$\boldsymbol{p}_2$ 和 $\boldsymbol{p}_1+\boldsymbol{p}_2=(x_1+x_2,\ y_1+y_2)$构成的平行四边形的带符号面积。一个等价的且更有用的定义是叉积是下列矩阵的行列式①：

$$\begin{aligned}\boldsymbol{p}_1\times\boldsymbol{p}_2&=\det\begin{pmatrix}x_1 & x_2\\ y_1 & y_2\end{pmatrix}\\&=x_1y_2-x_2y_1\\&=-\boldsymbol{p}_2\times\boldsymbol{p}_1\end{aligned}$$

图 5-1(a)所示为向量 $\boldsymbol{p}_1$ 和 $\boldsymbol{p}_2$ 的叉积，是平行四边形的带符号面积。图 5-1(b)中的浅阴影区域包含由向量 $\boldsymbol{p}$ 出发顺时针旋转的所有向量，而深阴影部分则包含从向量 $\boldsymbol{p}$ 出发反时针旋转的所有向量。

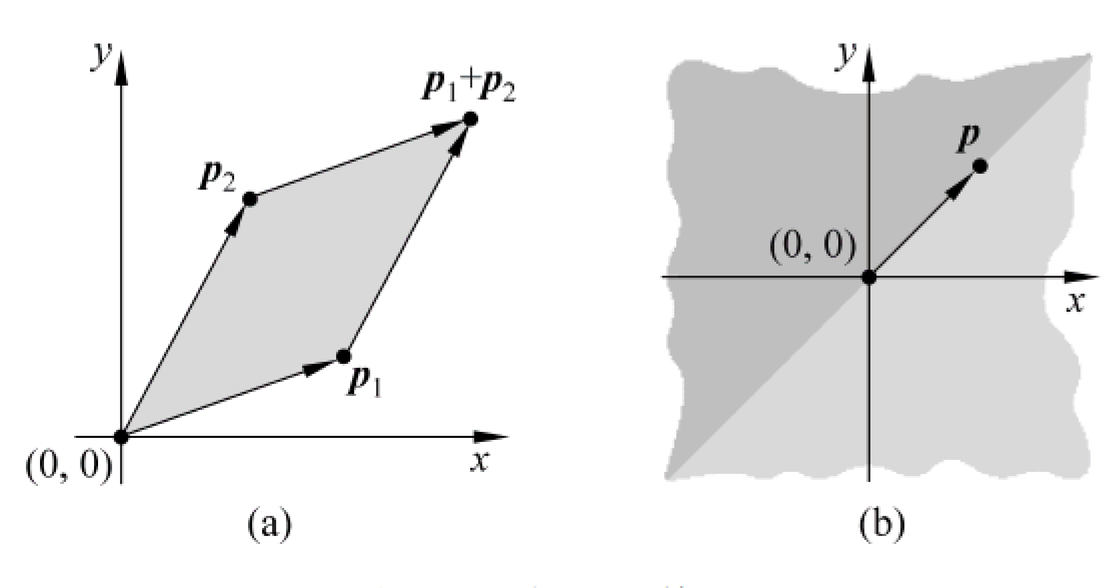


图 5-1　向量及其叉积

利用对向量叉积的定义，下列定理是很有用的。

定理 5-1　假定向量 $\boldsymbol{p}_1$ 和 $\boldsymbol{p}_2$ 所张角介于 0～π 之间，则：

(1) 若 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为负当且仅当 $\boldsymbol{p}_1$ 是从 $\boldsymbol{p}_2$ 反时针方向旋转而得；

(2) 若 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为正当且仅当 $\boldsymbol{p}_1$ 是从 $\boldsymbol{p}_2$ 绕原点(0,0)顺时针方向旋转而得；

(3) 若 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为零，则产生边界条件，此时，两个向量**共线**，或方向一致或方向相反。

图 5-2 形象地说明了定理 5-1。

图 5-2(a)所示向量 $\boldsymbol{p}_1$ 是从 $\boldsymbol{p}_2$ 反时针方向旋转而得。平行四边形 $Op_1p_3p_2$ 的面积＝三角形 Op_1p_1'面积＋梯形 $p_1p_3p_2'p_1'$－三角形 Op_2p_2'－梯形 $p_2'p_2p_3p_3'=x_1y_1/2+x_2(2y_1+y_2)/2-x_2y_2/2-x_1(2y_2+y_1)/2=x_2y_1-x_1y_2>0$。图 5-2(b)中向量 $\boldsymbol{p}_1$ 是从 $\boldsymbol{p}_2$ 反时针方向旋转而得。平行四边形 $Op_2p_3p_1$ 的面积＝梯形 $p_2'p_2p_3p_3'$面积＋梯形 $p_3'p_3p_1p_1'$－三角形 $p_2'p_2O$ 面积－三角形 $Op_1p_1'=-y_1(2x_2+x_2)/2+y_2(2x_1+x_2)/2-x_2y_2/2+x_1y_1/2=x_1y_2-x_2y_1>0$。图 5-2(c)向量 $\boldsymbol{p}_1$、$\boldsymbol{p}_2$ 共线。$y_2/x_2=y_1/x_1\Rightarrow x_2y_1-x_1y_2=0$。

利用定理 5-1，可直接判定有向线段$\overrightarrow{p_0p_1}$是否从有向线段$\overrightarrow{p_0p_2}$绕它们的公共端点 p_0 顺

① 事实上，叉积是一个三维概念。它是一个按“右手法则”既垂直于 $\boldsymbol{p}_1$ 又垂直于 $\boldsymbol{p}_2$ 的向量，其模长为$|x_1y_2-x_2y_1|$。然而，本章的内容说明，将叉积视为值 $x_1y_2-x_2y_1$ 是方便的。

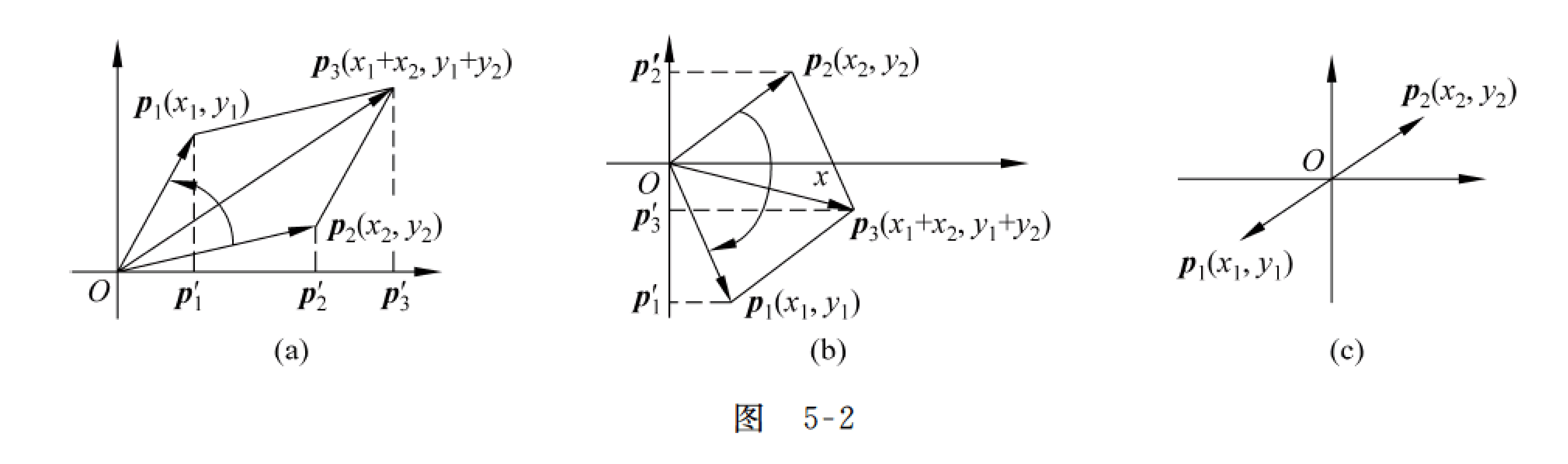


图 5-2

时针旋转而得。为此,只要将 $\boldsymbol{p}_0$ 转换为原点。即设 $\boldsymbol{p}_1-\boldsymbol{p}_0$ 表示向量 $\boldsymbol{p}'_1=(x'_1,y'_1)$,其中 $x'_1=x_1-x_0$ 及 $y'_1=y_1-y_0$。类似地,定义 $\boldsymbol{p}_2-\boldsymbol{p}_0$。然后计算叉积:

$$(\boldsymbol{p}_1-\boldsymbol{p}_0)\times(\boldsymbol{p}_2-\boldsymbol{p}_0)=(x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0)$$

如果此叉积为正,则 $\overrightarrow{p_0p_1}$ 是从有向线段 $\overrightarrow{p_0p_2}$ 绕它们的公共端点 p_0 顺时针旋转而得;若为负,则为逆时针方向。

2. 判断相继两直线段左转或右转

下一个问题是两条相继线段 $\overline{p_0p_1}$ 和 $\overline{p_1p_2}$ 是在点 p_1 处左转还是右转。等价地,要设法判定给定角 $\angle p_0p_1p_2$ 的转向。叉积使得我们可以不计算角而回答此问题。

```
DIRECTION(p0, p1, p2)                ▷计算 p0p1、p1p2 的转向
1 return (p2 − p0)×(p1 − p0)
```

算法 5-1 计算相继线段 $\overline{p_0p_1}$、$\overline{p_1p_2}$ 的转向

如图 5-3 所示,直接检测有向线段 $\overrightarrow{p_0p_2}$ 是从有向线段 $\overrightarrow{p_0p_1}$ 是顺时针还是逆时针旋转而得。为此,计算叉积 $(\boldsymbol{p}_2-\boldsymbol{p}_0)\times(\boldsymbol{p}_1-\boldsymbol{p}_0)$。若此叉积为负,则 $\overrightarrow{p_0p_2}$ 是从有向线段 $\overrightarrow{p_0p_1}$ 逆时针旋转而得的,因此在 p_1 处左转。正的叉积值意味着顺时针并右转。叉积为 0 意味着 p_0、p_1 和 p_2 共线。

图 5-3 所示为使用叉积判断相继线段 $\overline{p_0p_1}$、$\overline{p_1p_2}$ 在 p_1 处的转向。检测有向线段 $\overrightarrow{p_0p_2}$ 是从 $\overrightarrow{p_0p_1}$ 顺时针还是逆时针旋转而得。图 5-3(a)若是逆时针,则在该点处左转。图 5-3(b)若是顺时针,则为右转。

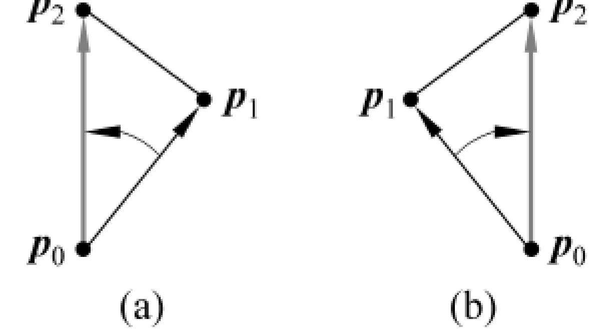


图 5-3 叉积及转向

3. 判断两条线段是否相交

为确定两条线段是否相交,检测每条线段是否跨越包含另一条线段的直线。线段 $\overline{p_1p_2}$ 的端点 p_1 位于一条直线的一边,而端点 p_2 位于另一边,则该线段**跨越**此直线。若 p_1 或 p_2 位于直线上则发生边界情形。两条线段相交当且仅当下列两个条件至少发生一个。

(1) 每一条线段跨越包含另一条线段的直线。

(2) 一条线段的一个端点位于另一条线段上。

下列过程实现了这一思想。若两条线段相交,SEGMENTS-INTERSECT 返回 TRUE,若不相交,返回 FALSE。它调用子过程 DIRECTION,该过程利用上述的叉积方法计算相对方

位，还要调用子过程 IN-BOX，该过程确定一个点是否落于以已知线段为对角线的矩形内。

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)　　▷检测线段$\overline{p_1p_2}$与$\overline{p_3p_4}$是否相交

1 $d_1 \leftarrow$ DIRECTION(p_3, p_4, p_1)　　▷计算$\overline{p_4p_3}$到$\overline{p_1p_3}$的转向

2 $d_2 \leftarrow$ DIRECTION(p_3, p_4, p_2)　　▷计算$\overline{p_4p_3}$到$\overline{p_2p_3}$的转向

3 $d_3 \leftarrow$ DIRECTION(p_1, p_2, p_3)　　▷计算$\overline{p_2p_1}$到$\overline{p_3p_1}$的转向

4 $d_4 \leftarrow$ DIRECTION(p_1, p_2, p_4)　　▷计算$\overline{p_2p_1}$到$\overline{p_4p_1}$的转向

5 **if** (($d_1 * d_2 < 0$) and ($d_3 * d_4 < 0$))

6 　　**then return** TRUE

7 **elseif** $d_1 = 0$ and IN-BOX(p_3, p_4, p_1)

8 　　**then return** TRUE

9 **elseif** $d_2 = 0$ and IN-BOX(p_3, p_4, p_2)

10 　　**then return** TRUE

11 **elseif** $d_3 = 0$ and IN-BOX(p_1, p_2, p_3)

12 　　**then return** TRUE

13 **elseif** $d_4 = 0$ and IN-BOX(p_1, p_2, p_4)

14 　　**then return** TRUE

15 **else return** FALSE

IN-BOX(p_i, p_j, p_k)　　▷检测点 p_k 是否处于以$\overline{p_ip_j}$为对角线的矩形内

1 **if** $\min(x_i, x_j) \leqslant x_k \leqslant \max(x_i, x_j)$ and $\min(y_i, y_j) \leqslant y_k \leqslant \max(y_i, y_j)$

2 　　**then return** TRUE

3 　　**else return** FALSE

算法 5-2　检测两条线段是否相交的算法

图 5-4 所示为过程 SEGMENTS-INTERSECT 中的各种情形。图 5-4(a)为线段$\overline{p_1p_2}$和$\overline{p_3p_4}$相互跨越对方所在直线。由于$\overline{p_3p_4}$跨越包含$\overline{p_1p_2}$的直线，叉积$(p_3-p_1)\times(p_2-p_1)$和$(p_4-p_1)\times(p_2-p_1)$的符号相反。由于$\overline{p_1p_2}$跨越包含$\overline{p_3p_4}$的直线，叉积$(p_1-p_3)\times(p_4-p_3)$和$(p_2-p_3)\times(p_4-p_3)$的符号相反。图 5-4(b)为线段$\overline{p_3p_4}$跨越包含$\overline{p_1p_2}$的直线，但$\overline{p_1p_2}$并不跨越包含$\overline{p_3p_4}$的直线。叉积$(p_1-p_3)\times(p_4-p_3)$和$(p_2-p_3)\times(p_4-p_3)$的符号相同。图 5-4(c)为点 p_3 与 p_1 和 p_2 共线且介于两者之间。图 5-4(d)为点 p_3 与$\overline{p_1p_2}$共线，但它并不介于 p_1 和 p_2 之间。两条线段不相交。

SEGMENTS-INTERSECT 运行如下。第 1～4 行计算每一个端点 p_i 关于其他线段的相对方位 d_i。若所有的相对方位非零，则不难确定两条线段$\overline{p_1p_2}$和$\overline{p_3p_4}$是否相交。若有向线段$\overrightarrow{p_3p_1}$和$\overrightarrow{p_3p_2}$具有相对于$\overrightarrow{p_3p_4}$的正的方位，线段$\overline{p_1p_2}$跨越包含线段$\overline{p_3p_4}$的直线。此时，d_1 和 d_2 的符号相反。相似地，若 d_3 和 d_4 的符号相反，则$\overline{p_3p_4}$跨越包含$\overline{p_1p_2}$的直线。若第 5 行的检测为真，则两线段相互跨越，且 SEGMENTS-INTERSECT 返回 TRUE。图 5-4(a)展示了此情形。否则，两条线段并不相互跨越对方所在的直线，但可能会发生边界情形。若所有的相对方位均非零，没有边界情形发生。若第 7～13 行的所有对 0 的测试都失败，则 SEGMENTS-INTERSECT 在第 15 行返回 FALSE。图 5-4(b)展示了这一情形。

任一相对方位 d_k 为 0，则发生边界情形。此时，知道 p_k 与另一条线段$\overline{p_ip_j}$共线。过程

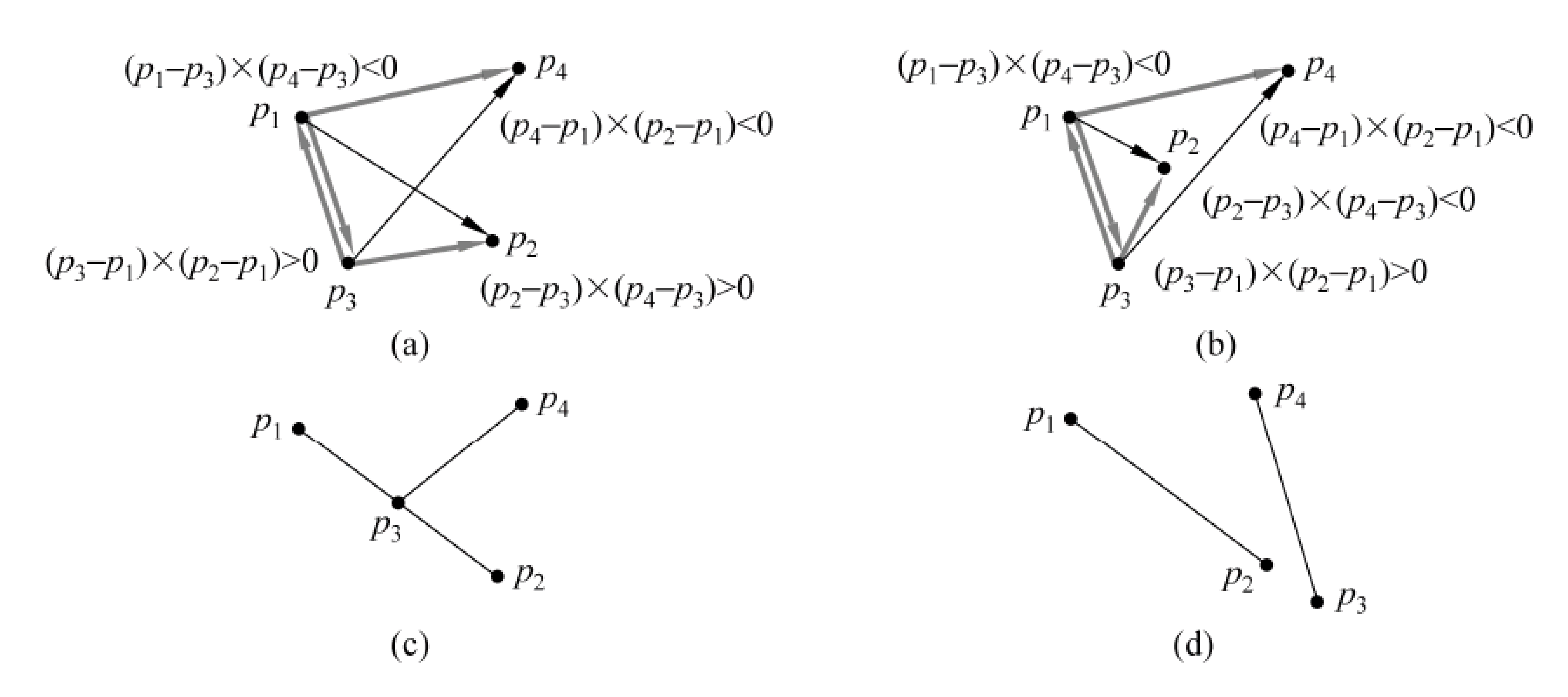


图 5-4　平面上两条线段的相交的各种情形

IN-BOX(p_i,p_j,p_k)返回 p_k 是否介于线段$\overline{p_ip_j}$的两个端点之间;该过程假定 p_k 与$\overline{p_ip_j}$共线。图 5-4(c)和图 5-4(d)展示了共线点的情形。在图 5-4(c)中,p_3 在$\overline{p_1p_2}$上,所以 SEGMENTS-INTERSECT 在第 12 行返回 TRUE。在图 5-4(d)中,没有端点在另一条线段上,所以 SEGMENTS-INTERSECT 在第 15 行返回 FALSE。

5.1.2　向量的极角

在一些几何算法中常用到极角的概念。把向量 p_1-p_0 与 x 正半轴的夹角称为点 p_1 关于点 p_0 的**极角**。例如,(3,5)关于(2,4)的极角是向量(1,1)与 x 正半轴的夹角 45°或 $\pi/4$ 弧度。(3,3)关于(4,2)的极角是向量(−1,1)与 x 正半轴的夹角 135°或 $3\pi/4$ 弧度。

算法中常需要比较两个向量极角的大小。数学中,要计算每个向量与 x 正半轴夹角的正切值$\frac{y-y_0}{x-x_0}$,然后利用反正切函数 arctan 以及向量所在的象限计算出极角。下面考虑一种不使用三角函数和反三角函数,而比较向量间极角大小的方法。不失一般性,设非零向量 $p(x,y)$的模$|p|$(p 到原点 O 的距离 $\sqrt{x^2+y^2}$)为 1,(如果$|p|\neq 1$,可按下述方法进行**规格化**:令 $p'=p/|p|=\left(\frac{x}{\sqrt{x^2+y^2}},\frac{y}{\sqrt{x^2+y^2}}\right)$,则 p' 与 p 方向一致,且$|p'|=1$。)记 p 与 x 正半轴的夹角为α。设 $p_0(1,0)$为 x 轴上的单位向量,记为(x_0,y_0),其与 x 轴正向夹角$\beta=0$。由于 p、p_0 都是规格化了的向量,所以 $x=\cos\alpha$,$y=\sin\alpha$,$x_0=\cos\beta$,$y_0=\sin\beta$。考虑叉积

$$\begin{aligned}p_0\times p &= yx_0-xy_0\\ &=\sin\alpha\cos\beta-\cos\alpha\sin\beta\\ &=\sin(\alpha-\beta)\\ &=\sin\alpha\end{aligned}$$

在数学中,单位圆上的弧长 α 较小时($0<\alpha<\pi/2$),$\sin\alpha\approx\alpha$。当 p 位于第Ⅰ象限时,如图 5-5(a)所示,就用 $y=\sin\alpha$ 替代 p 的极角 α。当 p 位于第Ⅱ象限时,如图 5-5(b)所示,$\sin\alpha=\sin(\pi/2+\alpha')$。此时,用 $\pi/2-x$ 替代 p 的极角 α。当 p 位于第Ⅲ象限时,如图 5-5(c)所示,$\sin\alpha=\sin(\pi+\alpha')$。此时,用 $\pi-y$ 替代 p 的极角 α。当 p 位于第Ⅳ象限时,如图 5-5(d)

所示，$\sin\alpha=\sin(3\pi/2+\alpha')$。此时，用 $3\pi/2+x$ 替代 p 的极角 α。

图 5-5 所示为规格化向量 p 与 $p_0=(1,0)$ 的叉积与极角的关系。图 5-5(a)中，p 位于第Ⅰ象限，$p_0\times p=\sin\alpha$。图 5-5(b)中，p 位于第Ⅱ象限，$p_0\times p=\sin\alpha=\sin(\pi/2+\alpha')$。图 5-5(c)中，$p$ 位于第Ⅲ象限，$p_0\times p=\sin\alpha=\sin(\pi+\alpha')$。图 5-5(d)中，$p$ 位于第Ⅳ象限，$p_0\times p=\sin\alpha=\sin(3\pi/2+\alpha')$。

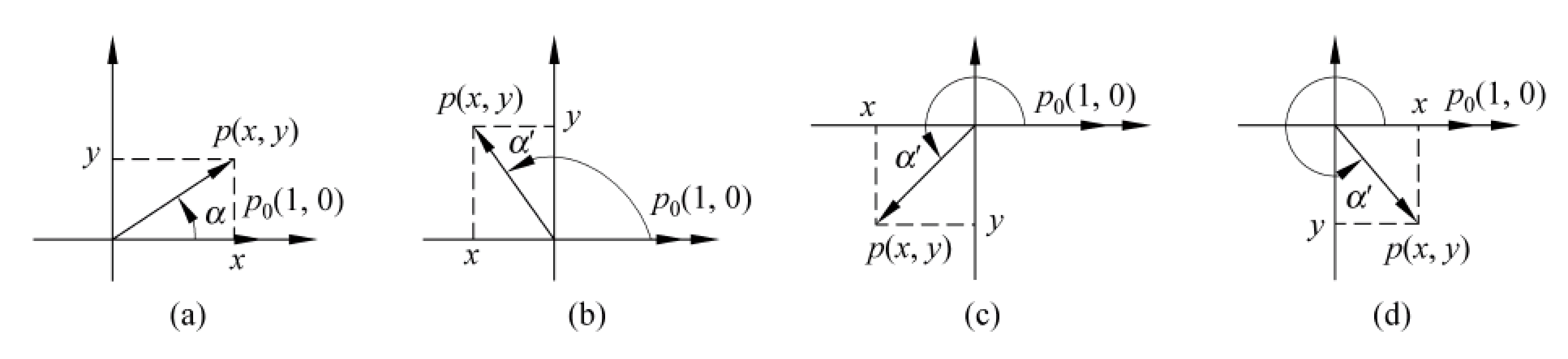


图 5-5　叉积与极角的关系

这样，无须真正计算出 p 的极角，却可以对两个规格化后的向量比较极角的大小。把上述替代极角的值称为向量的伪极角。下列过程描述了计算向量 p 相对于 p_0 的伪极角(即向量 $p-p_0$ 的伪极角)算法。

```
PSUDO-POLAR-ANGLE(p, p0)
p1(x1, y1) = p − p0
将 p1 规格化
if x1 ≥ 0 且 y1 ≥ 0              ▷第Ⅰ象限
    then return y1
if x1 < 0 且 y1 ≥ 0              ▷第Ⅱ象限
    then return π/2 − x1
if x1 < 0 且 y1 < 0              ▷第Ⅲ象限
    then return π − y1
return 3π/2 + x1                 ▷第Ⅳ象限
```

算法 5-3　计算向量 $p-p_0$ 的伪极角

5.1.3　程序实现

1. 平面中点的数据表示

实现平面上点的数据类型如下。

```c
#define epsilon 1e-10
#define PI 3.1415926
typedef struct{                                  /* 平面点 */
    double x,y;                                  /* 横坐标与纵坐标 */
}Point;
double dist(Point *a,Point *b){                  /* 两点间距离 */
    return sqrt(pow(a->x-b->x,2)+pow(a->y-b->y,2));
}
Point sub(Point *a,Point *b){                    /* 两点向量差 */
```

```
10      Point c={a->x-b->x,a->y-b->y};
11 return c;
12 }
13 double crossProduct(Point *a,Point *b){          /*叉积*/
14      return a->x*b->y-a->y*b->x;
15 }
```

程序 5-1　平面点数据类型及常规维护函数

程序 5-1 的说明如下。

(1) 第 3～5 行定义了表示平面上点的结构体类型 Point。Point 类型的数据具有 2 个 **double** 型的数据属性：点的横坐标 x 和纵坐标 y。

(2) 第 6～8 行定义的函数 dist 计算由参数 a、b 指引的两个 Point 点之间的距离。第 9～12 行定义的函数 sub 计算由参数 a、b 指引的两个 Point 点按坐标差计算所得的点(向量)。

(3) 第 13～15 行定义的函数 crossProduct 计算由参数 a、b 指引的两个 Point 点(向量)的叉积。

所有这些函数的定义代码都十分简单，读者不难阅读理解。Point 类型的定义及各函数的原型声明存储于文件夹 geometry 的头文件 point.h 中，函数的定义存储于同一文件夹的源文件 point.c 中。

2. 利用叉积检测线段性质

利用上述实现的平面上点的数据类型及向量的叉积计算函数，来实现算法 5-1 描述的 DIRECTION 过程和算法 5-2 描述的 SEGMENTS-INTERSECT 过程。

```
int direction(Point *p0,Point *p1,Point *p2){  /*计算向量 p2p0、p1p0 的夹角方向*/
    Point p=sub(p2,p0),q=sub(p1,p0);
    double d=crossProduct(&p,&q);
    if(d>0.0)                                   /*顺时针*/
        return 1;
    if(d<0.0)                                   /*逆时针*/
        return-1;
    return 0;                                   /*共线*/
}
int inBox(Point *pi,Point *pj,Point *pk){      /*检测 pk 落于以 pi、pj 为对角线的矩形内*/
    double x1=pi->x<pj->x?pi->x:pj->x,
        x2=pi->x>pj->x?pi->x:pj->x,
        y1=pi->y<pj->y?pi->y:pj->y,
        y2=pi->y>pj->y?pi->y:pj->y;
        return x1<=pk->x&&pk->x<=x2&&y1<=pk->y&&pk->y<=y2;
}
int segmentsIntersect(Point *p1,Point *p2,Point *p3,Point *p4){
    int d1=direction(p3,p4,p1),
        d2=direction(p3,p4,p2),
```

```
20          d3=direction(p1,p2,p3);
21          d4=direction(p1,p2,p4);
22      if(d1==0&&inBox(p3,p4,p1))
23          return 1;
24      if(d1==0&&inBox(p3,p4,p1))
25          return 1;
26      if(d2==0&&inBox(p3,p4,p2))
27          return 1;
28      if(d3==0&&inBox(p1,p2,p3))
29          return 1;
30      if(d4==0&&inBox(p1,p2,p4))
31          return 1;
32      return 0;
33 }
```

程序 5-2　实现算法 5-1 和算法 5-2 的 C 函数

对程序 5-2 的说明如下。

(1) 第 1～9 行定义的函数 direction 实现算法 5-1 的 DIRECTION 过程，计算由指针参数 p0、p1、p2 指引的 3 个点构成的两条连续线段 $\overline{p_0p_1}$ 和 $\overline{p_1p_2}$ 的转向。第 2 行调用程序 5-1 中定义的函数 sub，计算向量 p_2-p_0 和 p_1-p_0，分别赋予 p 和 q。第 3 行调用函数 crossProduct，计算叉积 $(p_2-p_0)\times(p_1-p_0)$ 并赋予 d。为便于使用，函数并不直接返回 d，而是根据 d 的符号返回 1 或 0 或 −1。

(2) 第 10～16 行定义的函数 inBox 实现算法 5-2 中 IN-BOX 过程，检验由指针参数 pk 指引的点 pk 是否落在由指针参数 pi、pj 指引的点连成的线段 $\overline{p_ip_j}$ 作为对角线的矩形内。其中，第 11～14 行计算 $\min(x_i,x_j)$、$\max(x_i,x_j)$、$\min(y_i,y_j)$ 和 $\max(y_i,y_j)$ 分别赋予 x1、x2、y1、y2。第 15 行检测条件 $\min(x_i,x_j)\leqslant x_k\leqslant\max(x_i,x_j)$ 且 $\min(y_i,y_j)\leqslant y_k\leqslant\max(y_i,y_j)$，并返回检测结果(条件成立返回 1 否则返回 0)。

(3) 第 17～33 行定义的函数 segmentsIntersect，实现算法 5-2 中的 SEGMENTS-INTERSECT 过程，检测由指针参数 p1、p2、p3 和 p4 指引的点构成的线段 $\overline{p_1p_2}$ 与 $\overline{p_3p_4}$ 是否相交。程序代码结构与算法的伪代码结构十分接近，读者可对比阅读，此处不再赘述。

3. 向量的伪极角

```
double pabs(Point *a){                          /*计算向量的模*/
    Point o={0.0,0.0};                          /*原点*/
    return dist(a,&o);
}
void normalize(Point *a){                       /*向量正规化(模长为 1)*/
    double r=pabs(a);
    if(r>=epsilon)                              /*非 0 向量*/
        a->x/=r,a->y/=r;
}
double psudoPolarAngle(Point *a,Point *b){      /*计算向量 a 关于向量 b 的极角*/
```

```
11      Point p1=sub(a,b);
12          normalize(&p1);
13      if(pabs(&p1)<epsilon)
14          return 2 * PI;
15      if(p1.y>=0.0&&p1.x>=0.0)            /* p 位于第Ⅰ象限 */
16          return p1.y;
17      if(p1.y>=0.0&&p1.x<0.0)             /* p 位于第Ⅱ象限 */
18          return PI/2-p1.x;
19      if(p1.y<0.0&&p1.x<0.0)              /* p 位于第Ⅲ象限 */
20          return PI-p1.y;
21      return 3 * PI/2+p1.x;               /* p 位于第Ⅳ象限 */
22 }
```

程序 5-3 实现算法 5-3 计算向量 a 关于向量 b 的极角的 PSUDO-POLAR-ANGLE 的函数

对程序 5-3 的说明如下。

(1) 第 1～4 行定义的函数 pabs 计算由指针参数 a 所指引的点到原点的距离——原点到该点形成的向量的模。

(2) 第 5～9 行定义的函数 normalize 对指针参数 a 指引的点构成的向量做规格化操作——使其模为 1。

(3) 第 10～22 行定义的函数 psudoPolarAngle 实现算法 5-3 的 PSUDO-POLAR-ANGLE 过程,计算由指针参数 a、b 指引的点构成的向量伪极角。程序代码结构与算法的伪代码结构十分接近,读者可对照阅读。

程序 5-2 和程序 5-3 中定义的各函数均存储于 geometry 文件夹的源文件 point.c 中,各函数的原型声明存储于同一文件夹的头文件中。

5.2 判断是否存在线段相交

判断线段是否相交的问题可如下形式化。

输入: 线段集合 $S=\{s_1, s_2, \cdots, s_n\}$。

输出: 如果 S 中存在两条线段相交,输出布尔值 TRUE,否则输出 FALSE。

本节介绍一个判断线段集合中任意两条线段是否相交的算法。该算法用到一种称为"扫描"的技术,该技术对很多计算几何算法都适用。该算法的运行时间是 $O(n\lg n)$,其中 n 是所给的线段数。算法仅判断是否有线段相交,并不输出任何具体的相交情形。

在**扫描**过程中,一条假想的**扫描线**从左向右扫过给定的几何对象集合。扫描线的运动方向是 x 轴,把它视为时间轴。通过将几何对象置于动态数据结构内,并利用对象间的关系,扫描提供了一种对几何对象的排序方法。本节的线段相交算法按从左到右的顺序考虑所有线段的端点并在每遇到一个端点时检测相交性。

为简化描述确定 n 条线段是否有相交情形的算法并证明其正确性,要做两个前提假设。首先,假设输入的线段没有垂直的。其次,假设没有三条线段相交于一点。

5.2.1　算法描述与分析

1. 线段排序

由于假设没有垂直的输入线段,任何输入线段与给定的垂直扫描线至多有一个交点。因此可以按线段与扫描线上的交点的纵坐标对线段排序。

更准确地说,考虑两条线段 s_1 和 s_2。如果在横坐标 x 处的垂直扫描线与两者均相交,则说此两线段在 x 处可比。若 s_1、s_2 在 x 处可比,且 s_1 与扫描线的交点高于 s_2 与扫描线的交点,则称在 x 处 s_1 在 s_2 之上,记为 $s_1>_x s_2$。例如,在图 5-6(a)中,有关系 $a>_r c$,$a>_t b$,$b>_t c$,$a>_t c$,以及 $b>_u c$。线段 d 与其他线段不可比。

对任意给定的 x,关系"$>_x$"是所有在 x 处与扫描线相交的线段的一个全序关系。在不同的 x 处,线段进入的序和离开的序可能是不同的。当线段的左端点遇到扫描线时进入序,而当右端点遇到扫描线时离开序。

当扫描线通过两条线段的交点时会发生什么呢? 如图 5-6(b)所示,它们的位置在全序中是相反的。扫描线 v 和 w 分处于线段 e 和 f 的交点的两边,有 $e>_v f$ 和 $f>_w e$。由于假定了没有三条线段共点,必有某扫描线 x 使得相交线段 e 和 f 在全序 $>_x$ 中是**挨着**的。任何通过图 5-6(b)的阴影区域的扫描线,如 z、e 和 f 在此全序中是紧挨着的。

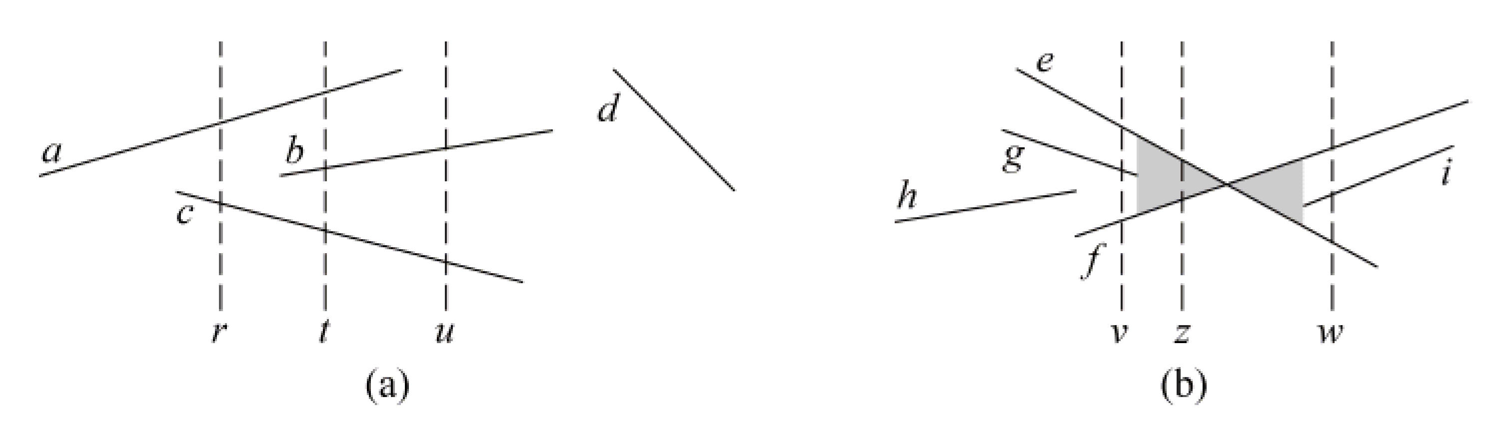


图 5-6　线段关于垂直扫描线的序

图 5-6 所示为线段关于垂直扫描线的序。图 5-6(a)中有 $a>_r c$,$a>_t b$,$b>_t c$,$a>_t c$,以及 $b>_u c$。线段 d 与其他线段不可比。图 5-6(b)中,当线段 e 和 f 相交时,它们的序是颠倒的:有 $e>_v f$ 和 $f>_w e$。任何通过阴影区域的扫描线(如 z)总使得 e 和 f 在全序中是紧挨着的。

2. 移动扫描线

扫描算法通常处理两个数据集合。

(1) **扫描线状态**给出与扫描线相交的对象间的关系。

(2) **事件点进度表**是一组按从左向右顺序横坐标序列,定义了扫描线的暂停位置,称这样的暂停位置为**事件点**。只有在事件点处才发生扫描线状态的改变。

在这个算法的事件点进度表中,每一个线段的端点为一个事件点。将各线段的端点按横坐标递增排序,从左向右逐一处理(若两个以上端点**共竖线**,即它们有共同的 x 坐标,按左端点先于右端点,对所有的共竖线左端点,按小 y 坐标先于大 y 坐标来打破僵局)。当遇到一条线段的左端点时,将其插入到扫描线状态,而当遇到该线段的右端点时,从扫描线状

态中将其删除。只要两条线段变为紧挨着的全序,就检测它们是否相交。

扫描线状态是一个全序 T,对其需要如下操作。

① INSERT(T,s):将线段 s 插入 T 中。

② DELETE(T,s):将线段 s 从 T 中删除。

③ ABOVE(T,s):返回 T 中直接位于 s 之上的线段。

④ BELOW(T,s):返回 T 中直接位于 s 之下的线段。

如果有 n 条输入线段,可以利用平衡树[①]在 $O(\lg n)$时间内执行以上的每一个操作。可以用叉积比较替代键值的比较来判断两条线段的相对序。

3. 伪代码描述

下列的算法将 n 条线段集合 S 作为输入,如果 S 中有相交线段则返回布尔值 TRUE,否则返回 FALSE。全序 T 用平衡树实现。

```
ANY-SEGMENTS-INTERSECT(S)
1  T←∅
2  对S 中的线段的端点从左到右排序
     按左端点先于右端点的方式打破僵局
     按小 y 坐标优先打破横坐标相等的僵局
3  for 排好序的端点列表中的每一个点 p
4      do if p 是一线段 s 的左端点
5            then INSERT(T,s)
6                 if (ABOVE(T,s)存在且与 s 相交)
                  or (BELOW(T,s) 存在且与 s 相交)
7                    then return TRUE
8         if p 是一线段 s 的右端点
9            then if ABOVE(T,s)和 BELOW(T,s)均存在
                  and ABOVE(T,s)与 BELOW(T,s)相交
10                   then return TRUE
11                DELETE(T,s)
12 return FALSE
```

算法 5-4　判断线段集合中是否有线段相交的算法

图 5-7 示例了该算法的执行。第 1 行初始化全序为空。第 2 行通过对 $2n$ 个线段端点从左到右的排序来确定事件点进度表,当端点的横坐标相等时按上述的方法打破僵局。注意第 2 行可以对(x,e,y)按字典方式排序,其中 x 和 y 是通常的坐标,而 $e=0$ 表左端点,$e=1$ 表右端点。

第 3～11 行的 **for** 循环每一次重复处理一个事件点 p。如果 p 是线段 s 的左端点,第 5 行把 s 加入到全序中去,若由通过 p 的扫描线定义的全序中紧挨 s 的两条线段中有一条与之相交,第 6 行和第 7 行返回 TRUE(若 p 落在另一条线段 s'上,发生一个边界条件。此时,

① 平衡树指的是左右子树的高度相差渐进常数的二叉搜索树,其中的元素按中序排列是有序的。例如,红-黑树就是这样的平衡树。

我们只要将 s 与 s' 相继置于 T 中)。如果 p 是线段 s 的右端点,则将 s 从全序中删除。若由通过 p 的扫描线定义的全序中紧挨 s 的两条线段中有一条与之相交,第 9 行和第 10 行返回 TRUE。这两条线段在 s 被删除后而在 T 中相邻。如果这些线段不相交,第 11 行将 s 从 T 中删除。最后,若处理完了 $2n$ 个事件点后,没有找到交点,第 12 行返回 FALSE。

图 5-7 所示为 Any-Segments-Intersect 的执行。每一根虚线为事件点处的扫描线,每条扫描线下是 **for** 循环对应于每个事件点处理的末尾的全序 T 中的线段名。当删除线段 c 时,找到线段 d、b 相交。

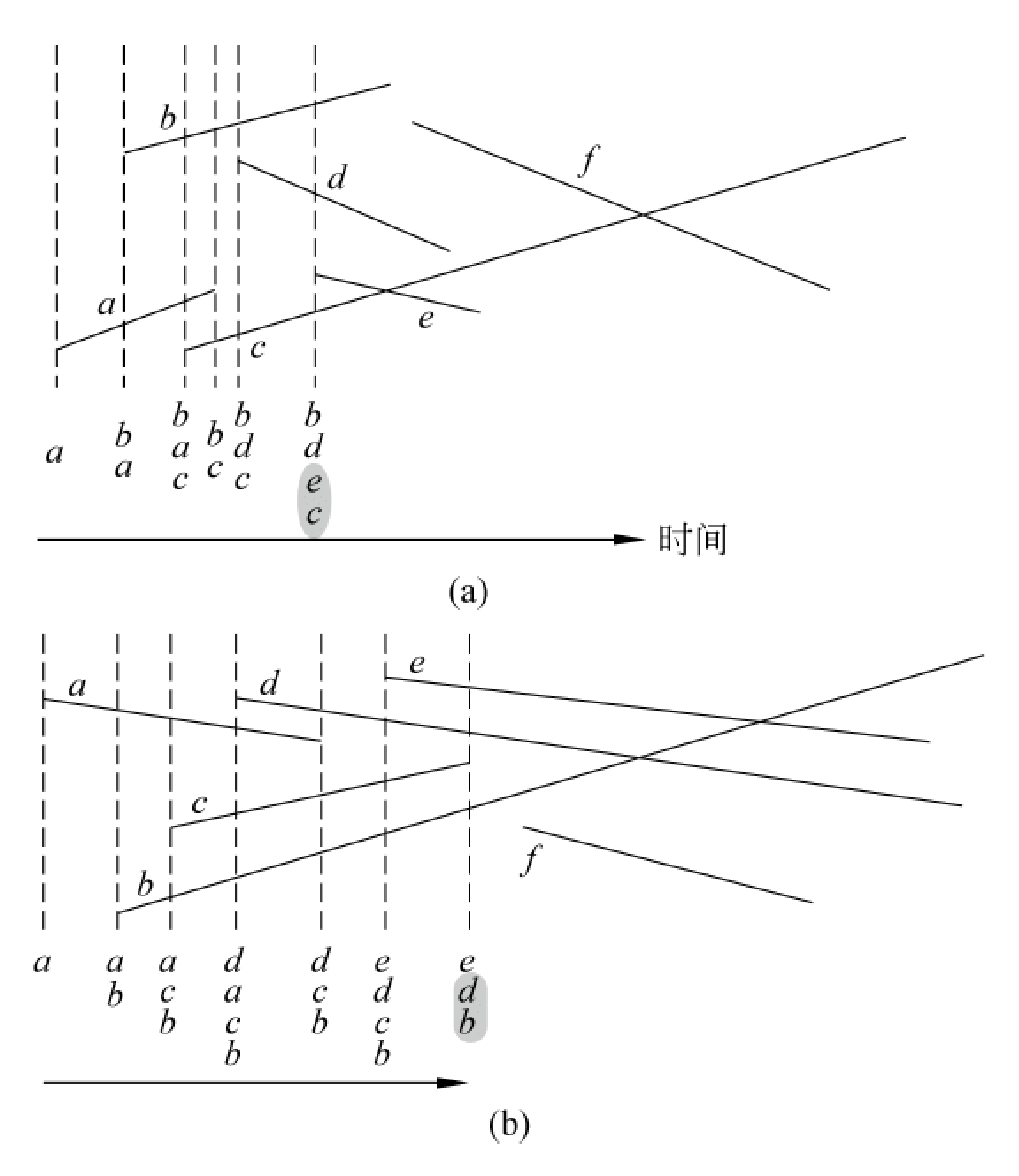


图 5-7　Any-Segments-Intersect 的执行

4. 算法的正确性

用定理 5-2 来说明 Any-Segments-Intersect 过程的正确性。

定理 5-2　调用 Any-Segments-Intersect(S)返回 TRUE 当切仅当 S 中有线段相交。

证明　过程 Any-Segments-Intersect 不正确只有在不存在相交的情况下返回 TRUE 或有相交线段却返回 FALSE。前者是不可能发生的,这是因为 Any-Segments-Intersect 只有在找到两条线段的相交时才返回 TRUE。

用反证法来证明后者。假设 S 中有线段相交,但 Any-Segments-Intersect 返回 FALSE。设 p 是最左端的交点,且线段 a 和 b 相交于 p。由于在 p 的左边没有交点发生,由 T 给出的序在 p 的左边都是恰当的。因为没有三条线段共点,存在扫描线 z,使得线段 a 和 b 在全序中是紧挨着的。此外,z 在 p 的左边或正通过 p。存在某条线段的端点 q 在扫描线 z,即 q 是一个事件点,在此处,a 和 b 在全序中变成紧挨着的。若 p 在扫描线 z 上,则 $p=q$。若 p 不在扫描线 z 上,则 q 在 p 的左边。无论哪种情形,由 T 给出的全序在 q 处理以前是正确的。在事件点 q 处,只可能发生两个动作之一。

(1) a 或 b 插入 T,并且有另一条线段在其上或其下。第 4~7 行能测得它。

(2) 线段 a 和 b 已经在 T 中,并且它们之间的线段已经在全序中被删除,使得 a 和 b 是紧挨着的。第 8~11 行测得此情形。

无论哪种情形,交点 p 均被找到,此与过程返回 FALSE 的假设矛盾。

5. 运行时间

若集合 S 中有 n 条线段,则 ANY-SEGMENTS-INTERSECT 的运行时间为 $O(n\lg n)$。第 1 行耗时 $O(1)$,第 2 行使用合并排序或堆排序耗时 $O(n\lg n)$。由于有 $2n$ 个事件点,第 3~11 行的 **for** 循环至多重复 $2n$ 次。由于每个平衡树操作耗时 $O(\lg n)$并且是用第 5.1 节中的方法,每个交点的测试耗时 $O(1)$,故每次重复耗时 $O(\lg n)$。总耗时为 $O(n\lg n)$。

5.2.2 程序实现

1. 线段的数据表示

线段由两个端点确定:左端点和右端点。显然线段端点是平面上的点,这只要在 Point 类的基础上添加一个表示端点类别(左端点/右端点)的属性 e 就可以了。

```
typedef struct{
    Point point;                              /* 端点坐标 */
    int e;                                    /* 端点标志(左端点 e=0,右端点 e=1) */
}EndPoint;                                    /* 线段端点类型 */
int pointLess(EndPoint *a,EndPoint *b){       /* 端点比较 */
    if(a->point.x<b->point.x)
        return 1;
    if(a->point.x==b->point.x)
        if(a->e==b->e)
            return a->point.y<b->point.y;
        else
            return a->e<b->e;
    return 0;
}
typedef struct{
    EndPoint left;                            /* 左端点 */
    EndPoint right;                           /* 右端点 */
    double length;                            /* 长度 */
    double tan;                               /* 斜率 */
    double yOffset;                           /* 所在直线在 y 轴的截距 */
}Segment;                                     /* 线段类 */
Segment newSeg(Point *a,Point *b){            /* 生成线段 */
    Segment c;
    c.left.point=*a;
    c.left.e=0;
    c.right.point=*b;
```

```
27      c.right.e=1;
28      c.length=dist(&c.left.point,&c.right.point);
29      c.tan=((c.left.point).y-(c.right.point).y)/((c.left.point).x-(c.right.point).x);
30      c.yOffset= c.left.point.y-c.tan*(c.left.point).x;
31      return c;
32 }
```

程序 5-4　线段类型的定义

对程序 5-4 的说明如下。

(1) 第 1～4 行将线段的端点定义为结构体 EndPoint。其中 point 成员是程序 5-1 中定义的平面点 Point 类型,记录端点的坐标。**int** 型成员 e 用来标识端点类型。约定:左端点 e=0,右端点 e=1。

(2) 第 5～14 行定义的函数 pointLess 比较由参数 a、b 指引的两个端点的“大小”。比较的规则是按端点的横坐标、类型和纵坐标构成的三元组(x,e,y)的字典顺序。也就是说,先比较两个点的横坐标:先小后大;若遇到两个点的横坐标相等的僵局,则按端点类型先左后右规则加以比较;若遇上两个点同时为两条线段的左端点或右端点的僵局,则按纵坐标先小后大加以比较。

(3) 第 15～21 行将线段定义为结构体 Segment。第 22～32 行定义的函数 newSeg 用参数 a、b 指引的两个点初始化一条线段。其中,第 24～27 行将线段的左、右端点初始化为 a、b。第 28 行调用程序 5-1 中定义的函数 dist,计算表示线段长度的成员 length。第 29 行计算表示线段所在直线的斜率的成员 tan,这只要直接计算两个端点的纵坐标之差与横坐标之差的商就行了。第 30 行计算表示线段所在直线在 y 轴上的截距的成员 yOffset。由于已经知道了该直线的斜率 tan 和过左端点 (x_1, y_1),所以该直线方程为 $y-y_1=\tan(x-x_1)$。为计算直线在 y 轴上的截距,只要在此方程中令 $x=0$,算出 y 的值就行了,如图 5-8 所示。

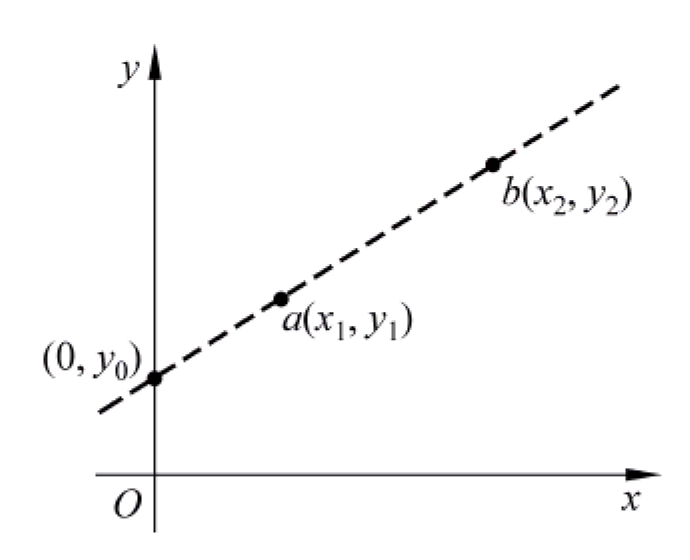


图 5-8　线段 ab 所在直线在 y 轴上的截距

2. 事件点进度表与扫描线状态表

用事件扫描的方法检测线段集合 S 中是否有线段相交,需要维护两个集合:事件点进度表 p 和扫描线状态表 T。事件点进度表(可以用数组来表示)中作为事件点的线段端点需记录它所在的线段——线段在数组 S 中的下标。而加入到扫描线状态表中的线段要组织成一棵平衡树,用 2.2.4 节的程序 2-16 至程序 2-32 中开发的红-黑树来表示。由于要对事件进度表进行排序预处理,所以需要利用程序 5-4 中定义的端点比较函数 pointLess 来定义事件点的比较规则。而要将线段加入到扫描线状态表 T,需要定义线段的全序比较规则。

```
typedef struct{                         /* 事件点类型 */
    EndPoint point;                     /* 端点 */
    int index;                          /* 端点所在线段在数组 S 中的下标 */
}p_i;
```

```
5  p_i newp_i(EndPoint p,int i){                  /* 生成事件点表中的元素 */
6      p_i x;
7      x.point=p;
8      x.index=i;
9      return x;
10 }
11 int p_iComp(p_i *a,p_i *b){                    /* 按端点的比较规则比较事件点 */
12     return pointLess(&(a->point),&(b->point));
13 }
14 double getY(Segment *a,double x){              /* 计算线段在 x 处的纵坐标 */
15     assert(x>=a->left.point.x&&x<=a->right.point.x);
16     return a->tan*x+a->yOffset;
17 }
18 int segComp(Segment *a,Segment *b){            /* 线段比较 */
19     double x0=a->left.point.x>b->left.point.x?a->left.point.x:b->left.point.x,
20         x1=a->right.point.x<b->right.point.x?a->right.point.x:b->right.point.x,
21         x=(x0+x1)/2.0;
22     return getY(a,x)>getY(b,x);
23 }
```

程序 5-5　事件点类型定义及线段比较规则

对程序 5-5 的说明如下。

(1) 第 1～4 行将本问题的事件点定义为结构体 p_i。它是在 EndPoint 类型的基础上添加一个表示端点所在线段在数组 S 中的下标的成员 index。第 5～10 行定义的函数 new p_i用参数端点 p 及参数 i 初始化新建的事件点。

(2) 第 11～13 行定义的函数 p_iComp 对由参数 a、b 指引的事件点对应的端点,调用程序 5-4 中定义的函数 pointLess 比较它们的"大小"。该函数用来对事件点进度表进行排序操作。

(3) 第 18～23 行定义的函数 segComp 按正文中讨论的规则比较由参数 a、b 指引的两条线段的关系。第 19 行和第 20 行分别计算 x1=max(a 的左端点,b 的左端点),x2=min(a 的右端点,b 的右端点),第 21 行计算 x1、x2 的平均值为 x。第 22 行调用函数 getY,比较线段 a 和 b 在 x0 处的纵坐标,并借以比较两者的上、下关系。函数 getY 定义于第 14～17 行,该函数计算由参数 a 指引的线段在参数 x 处的纵坐标。对 a 所在的直线,已知斜率为 tan,y 轴上的截距为 yOffset,则在 x 处的纵坐标为 a->tan * x+a->yOffset。

3. 判断线段集合 *S* 中是否有相交线段

有了程序 5-4 和程序 5-5 的准备,下面来实现算法 5-4 的 ANY-SEGMENTS-INTERSECT 过程,检测线段集合 S 中是否有线段相交。

```
int anySegmentIntersect(Segment* S,int n){
  int i;
  RBTree *T=creatRBTree(sizeof(Segment),segComp);   /* 扫描线状态表 */
  p_i    *p=(p_i*)malloc(2*n*sizeof(p_i));          /* 事件点进度表数组 */
```

```
    for(i=0;i<n;i++){                              /* 填写事件进度表 */
        p[2*i]=newp_i(getLeft(&S[i]),i);
        p[2*i+1]=newp_i(getRight(&S[i]),i);
    }
    quickSort(p,sizeof(p_i),0,n-1,p_iComp);        /* 对事件进度表排序 */
    for(i=0;i<2*n;i++){                            /* 对事件点进度表扫描 */
        Segment *s=&S[p[i].index],*above,*below;
        RBNode* pt;
        Point p1,p2,p3,p4;
        if(p[i].point.e==0){                       /* 如果该端点为某线段的左端点 */
            p1=getLeft(s).point;p2=getRight(s).point;
            pt=rbInsert(T,s);                      /* 将线段 s 插入 T,pt 指向该线段 */
            above=(Segment*)(treePredecessor(pt))->key;
            below=(Segment*)(treeSuccessor(pt))->key;
            if(above!=NULL){                       /* above 是紧挨着 s 且处于上方的线段 */
                p3=getLeft(above).point;p4=getRight(above).point;
                if(segmentsIntersect(&p1,&p2,&p3,&p4))     /* above、s 相交 */
                    break;
            }
            if(below!=NULL){                       /* below 是紧挨着 s 且处于下方的线段 */
                p3=getLeft(below).point;p4=getRight(below).point;
                if(segmentsIntersect(&p1,&p2,&p3,&p4))     /* s、below 相交 */
                    break;
            }
        }else{                                     /* 若该端点是某线段的右端点 */
            pt=rbSearch(T,s);                      /* pt 指向线段 s */
            above=(Segment*)(treePredecessor(pt))->key;
            below=(Segment*)(treeSuccessor(pt))->key;
            if(above!=NULL&&below!=NULL){
                p1=getLeft(above).point;p2=getRight(above).point;
                p3=getLeft(below).point;p4=getRight(below).point;
                if(segmentsIntersect(&p1,&p2,&p3,&p4))     /* above、below 相交 */
                    break;
            }
            rbDelete(T,pt);                                /* 从 T 中删除该线段 */
        }
    }
    free(p);
    clrRBTree(T,NULL);free(T);
    if(i<2*n)                                              /* 有线段相交 */
        return 1;
    return 0;                                              /* 没有相交线段 */
}
```

程序 5-6　实现算法 5-4 中 ANY-SEGMENTS-INTERSECT 过程的 C 函数

对程序 5-6 的说明如下。

(1) 函数 anySegmentIntersect 有两个参数,表示线段集合的数组 S 和 S 中所含线段数 n。若 S 中有线段相交,该函数返回 1,否则返回 0。

(2) 第 3 行定义了红-黑树 RBTree 型指针 T,并为其生成一棵空树,决定树中结点前后关系的规则为程序 5-5 中定义的函数 segComp。第 4 行为指向事件点进度表的 p_i 型指针 p 分配能存储 2n 个元素的数组空间,第 5~8 行将 S 中 n 条线段的左右端点填写到 p 指向的数组中。第 9 行调用 3.2.2 节程序 3-8 开发的快速排序函数 quickSort,按程序 5-5 中定义的比较线段端点构成的事件点先后关系的函数 p_iComp 对数组 p 排序,从而确定事件点进度表。

(3) 第 10~41 行的 **for** 循环实现算法中第 3~11 行的 **for** 循环。程序代码与算法伪代码的结构十分接近。需要说明的是,对计算扫描状态表 T 中与线段 s 紧挨着且处于上方线段的 ABOVE 过程,由于使用第 2 章中开发的红-黑树表示 T,这相当于调用 treePredecessor 函数在 T 中计算 s 的前驱。类似地,调用 treeSuccessor 函数,计算 T 中 s 的后继来实现算法中的 BELOW 过程的调用,且从方便起见,将这两个函数调用的返回值赋予变量 above 和 below(见第 17、18、31、32 行)。在检测 above 与 s、below 与 s、above 与 below 是否相交时,调用程序 5-2 中定义的函数 segmentsIntersect(见第 21 行和第 36 行)。此外,为防止内存泄漏,函数返回前第 42 和第 43 行负责释放指针 p 指向的动态存储空间和红-黑树 T 的存储空间。

程序 5-4 至程序 5-6 中定义的数据类型和函数的原型声明存储于文件夹 geometry 中的头文件 segment.h 中,各函数的定义存储于同一文件夹中的源文件 segment.c 中。

5.3 求凸壳

点的集合 Q 的**凸壳**是一个凸多边形 P,Q 中的每个点要么在 P 的边界上,要么在 P 的内部。我们用 CH(Q)表示 Q 的凸壳。直观地,把 Q 中的点看成钉在板上的钉子。Q 的凸壳看成是用橡筋箍在这些钉子上围成的形状。图 5-9 展示了一个点的集合及其凸壳。

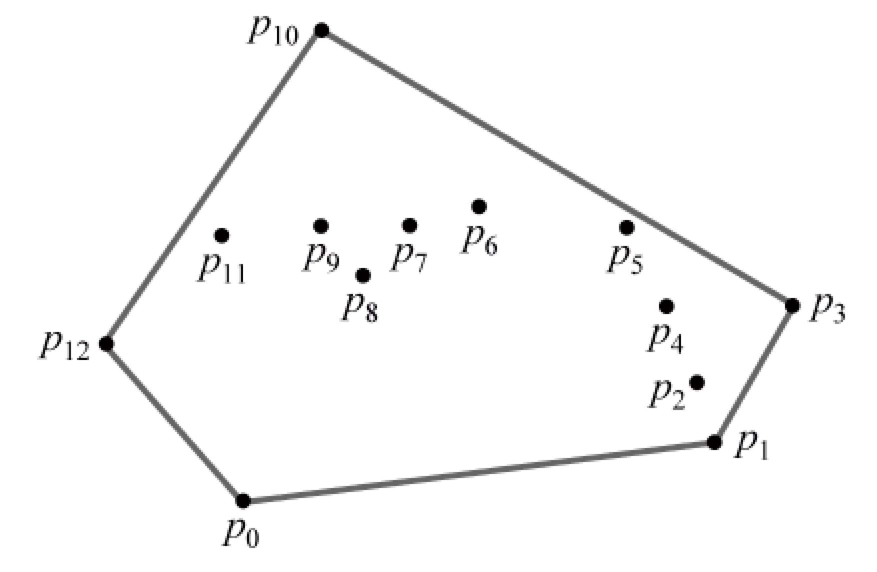


图 5-9 点集合 $Q=\{p_0,p_1,\cdots,p_{12}\}$ 及其灰色的凸壳 CH(Q)

在本节中,将介绍两个算法来计算 n 个点的集合的凸壳。这两个算法都按逆时针方向输出凸壳的顶点。第一个算法称为 Graham 扫描,运行时间为 $O(n\lg n)$;第二个算法称为 Jarvis 行进,运行时间为 $O(nh)$,其中 h 为凸壳的顶点数。如图 5-8 所示,CH(Q)的每一个顶点都是 Q 的点。两个算法都使用这一特性,决定 Q 中的哪些顶点作为凸壳的顶点加以保留,而舍弃哪些顶点。

计算点集的凸壳本身是一个很有趣的问题。此外,一些其他的计算几何问题以计算凸壳为开始。例如,考虑二维的**最远点对**问题:平面上给定 n 个点,希望找到其中相距最远的两个点。这两个点必是凸壳边界上的顶点,尽管我们在此并不证明。事实上,n 个顶点的凸壳的最远点对可以在 $O(n)$时间内求得。于是,通过在时间 $O(n\lg n)$内计算出 n 个输入点的

凸壳，然后求得所得的凸壳的最远点对，可以在时间 $O(n\lg n)$ 内找到 n 个点的集合的最远点对。

5.3.1　Graham 扫描

1. 算法描述

Graham 扫描法通过维护一个候选点的栈 S 来解决凸壳问题。输入集合 Q 的每一个点被压入栈一次，并且不是 CH(Q)顶点的点最终将被弹出栈 S。在算法终结时，栈 S 恰好包括了 CH(Q)的顶点，并以它们在边界上出现的逆时针顺序排列。

过程 GRAHAM-SCAN 将点集 Q 作为输入，其中 $|Q|\geqslant 3$。它调用函数 TOP(S)，该函数返回处于栈 S 的栈顶的点而不改变 S。调用函数 NEXT-TO-TOP(S)，它返回栈顶下的点而不改变栈 S。下面就要证明由 GRAHAM-SCAN 返回的栈 S 从栈底到栈顶恰好是 CH(Q)的顶点按逆时针的排列。

```
GRAHAM-SCAN(Q)
1  设 p0 是 Q 中最小纵坐标的点，多个这样的点取最左边的点
2  设<p1, p2, …, pm>为 Q 中剩下的其余点，并按关于 p0 的极角逆时针顺序排列
   （若有多个极角相同的点，留取其中距 p0 最远的点而删除其余点）
3  PUSH(p0, S)
4  PUSH(p1, S)
5  PUSH(p2, S)
6  for i ← 3 to m
7      do while 由点 NEXT-TO-TOP(S)、TOP(S)和 pi 构成的角非左转
8          do POP(S)
9             PUSH(pi, S)
10 return S
```

算法 5-5　计算平面点集凸壳的算法过程 GRAHAM-SCAN

图 5-10 示例了 GRAHAM-SCAN 的执行。第 1 行把具有最小纵坐标的点选为 p_0。对多个具有如此相同纵坐标的点，取横坐标最小的。由于在 Q 中不存在 p_0 以下的点，且具有相同纵坐标的点都处于 p_0 的右边，所以 p_0 是 CH(Q)的顶点。第 2 行利用算法 5-3 中伪极角的方法把 Q 中其余的点按关于 p_0 的极角排序。如果两个或多个点具有相同的关于 p_0 的极角，则除了最远的那一个以外，所有的都是 p_0 与那个最远点的凸组合，因此将它们都一并舍去。用 m 表示剩余的不同于 p_0 的点的数目。极角用弧度度量，Q 中的每一个点关于 p_0 的极角在半闭半开区间$[0,\pi)$内。由于极角的增加的方向是逆时针的，所以存储的点也是关于 p_0 逆时针顺序的。用$<p_1,p_2,\cdots,p_m>$表示这一排好序的序列，它们是 CH(Q)的顶点。图 5-10(a)展示了图 5-9 的那些点，并按关于 p_0 的极角递增顺序编号。

图 5-10 所示为对图 5-9 中的集合 Q 执行 GRAHAM-SCAN。包含在栈 S 的当前凸壳在每一步中用灰色连线显示。图 5-10(a)中，$<p_1,p_2,\cdots,p_{12}>$是按关于 p_0 的极角排列的点的序列。栈的初始状态含有 p_0、p_1 和 p_2。图 5-10(b)～图 5-10(k)对应第 6～9 行的 **for** 循环的每一次重复后的栈 S。虚线表示的是非左转的点，而被弹出栈 S。例如在图 5-10(h)

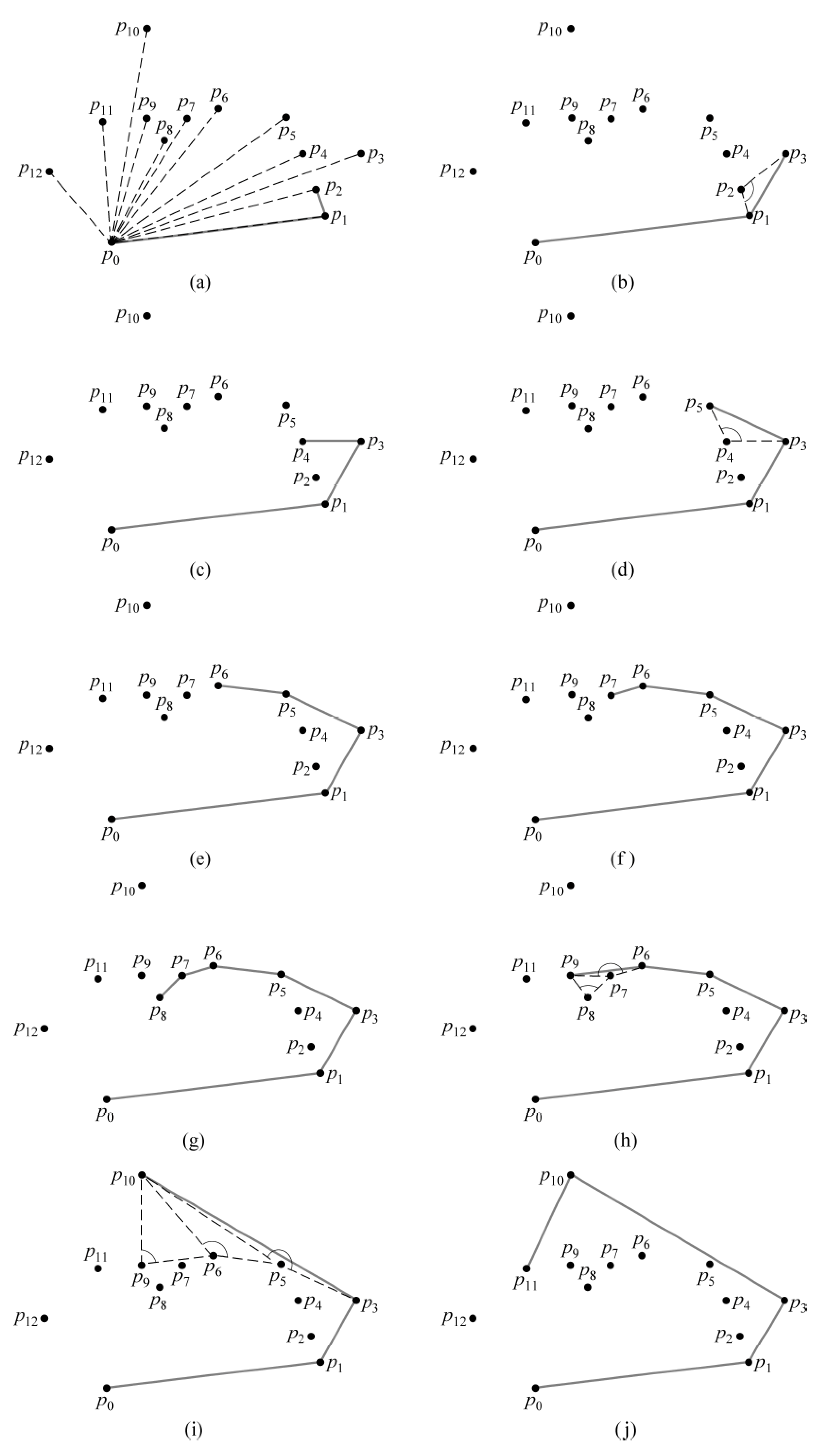


图 5-10　对图 5-9 中的集合 Q 执行 GRAHAM-SCAN

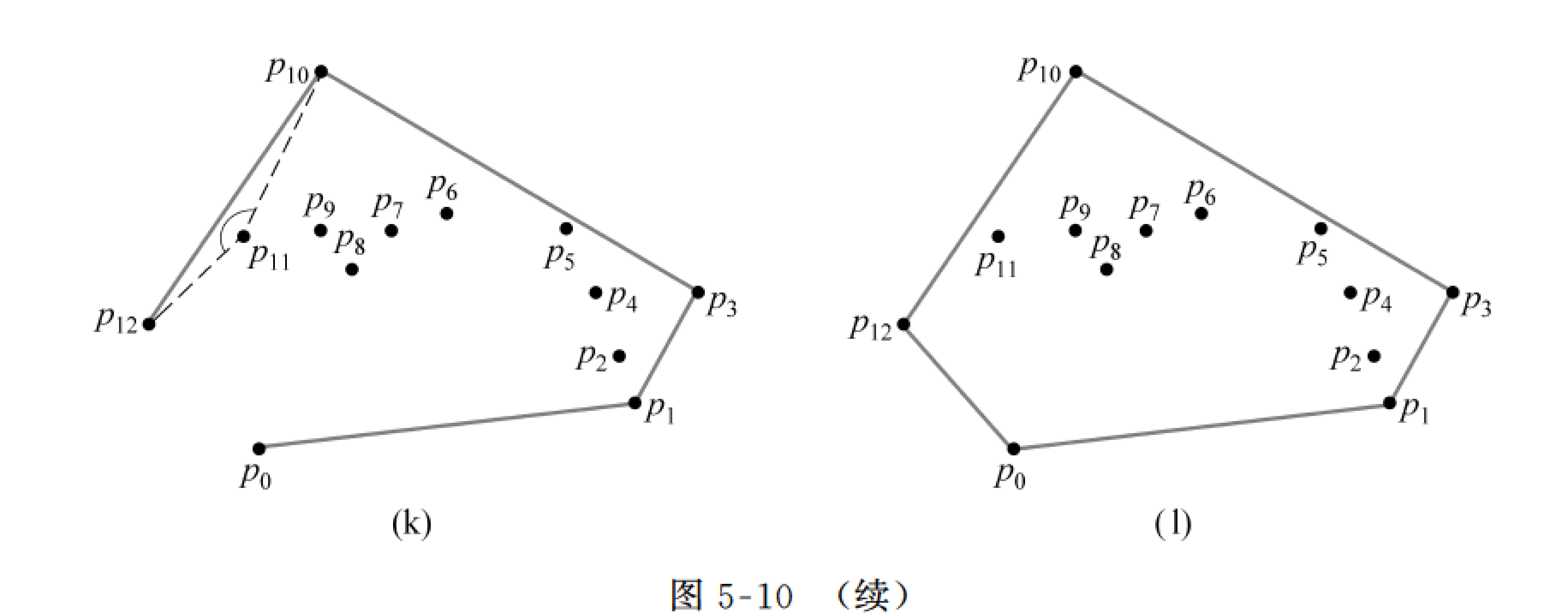


图 5-10　(续)

中，右转角$\angle p_7p_8p_9$ 导致 p_8 被弹出，然后右转角$\angle p_6p_7p_9$ 导致 p_7 被弹出。由过程 GRAHAM-SCAN 返回的凸壳与图 5-9 相符。

过程其余部分使用栈 S。第 3～6 行初始化栈，令其从栈底到栈顶包含 p_0、p_1 和 p_2。图 5-10(a)展示了初始的栈 S。第 6～9 行的 **for** 循环对序列$<p_3, p_4, \cdots, p_m>$中的每一个点重复一次。目的是使在处理完点 p_i 后栈 S 中从底到顶按逆时针方向的顺序包含了 CH($\{p_0, p_1, \cdots, p_i\}$)的顶点。第 7～8 行的 **while** 循环把发现的不是凸壳顶点的点从栈中去掉。当我们按逆时针方向遍历凸壳顶点的时候，在每个顶点处总是左转弯。因此，在此 **while** 循环中每次遇到非左转弯的顶点就把它从栈中弹出(检测非左转而不是右转，排除了凸壳上的平角可能性。这是我们所希望的，因为凸壳的任一顶点都不应是该多边形另外两顶点的凸组合)。在弹出所有以 p_i 为首的非左转顶点后，将 p_i 压栈。图 5-10(b)～图 5-10(k)展示了 **for** 循环的每一次重复后栈 S 的状态。最后，GRAHAM-SCAN 在第 11 行返回栈 S。图 5-10(l)展示了逆时针的凸壳。

2. 算法正确性

下列定理形式地证明了 GRAHAM-SCAN 的正确性。

定理 5-3(Graham 扫描的正确性)　如果 GRAHAM-SCAN 运行于点集 Q，其中$|Q| \geq 3$，则 Q 中的点在终结时在栈 S 中从底至顶恰含有 CH(Q)的按逆时针方向的各顶点。

证明　第 2 行后，有点的序列$<p_1, p_2, \cdots, p_m>$。对 $i=2,3,\cdots,m$，定义点的子集合 $Q_i=\{p_0, p_1, \cdots, p_i\}$。在 $Q-Q_m$中的点是那些因为与 Q_m 中的点具有相同的关于 p_0 的极角而被删除的点；这些点不在 CH(Q)中，所以 CH(Q_m)＝CH(Q)。于是，只要说明在 GRAHAM-SCAN 终止时，栈 S 从底到顶由 CH(Q_m)的各顶点按逆时针方向组成。注意如同 p_0、p_1 和 p_m是 CH(Q)的顶点，p_0、p_1 和 p_i 是 CH(Q_i)的顶点。

证明用到如下循环不变量。

在第 6～9 行的 **for** 循环的每一次重复开始时，栈 S 从底到顶由 CH(Q_{i-1})的顶点按逆时针方向组成。

对 Q 中的顶点数 i 作归纳。首次执行第 6 行时 $i=3$，不变量是成立的。这是因为此时栈 S 恰由 $Q_2=Q_{i-1}$的顶点组成，且此集合的 3 个顶点自身构成一个凸壳。此外，它们按出现的逆时针方向自底向顶存于 S 中。设 $3<i<n$ 时循环不变量是正确的。进入 **for** 循环的此次重复，S 的栈顶点是 p_{i-1}，它是在上一次重复结束时压入栈的(当 $i=3$ 时，是在第一次

重复前)。设 p_j 是执行了第 7 行和第 8 行的 **while** 循环后而在第 9 行压入 p_i 之前 S 的栈顶的点,且 p_k 是 S 中恰在 p_j 之下的点。在 p_i 尚未压入,p_j 为 S 的栈顶点之时,栈 S 恰包含了 **for** 循环的第 j 次重复以后 S 中所包含的点。按循环不变量,此时 S 恰包含 $CH(Q_j)$的顶点,并从底至顶按逆时针方向顺序出现。

我们继续关注 p_i 压栈前的时刻。参考图 5-10(a),由于 p_i 关于 p_0 的极角大于 p_j 的极角,且角$\angle p_k p_j p_i$ 左转(否则的话将弹出 p_j),我们看到由于 S 恰包含了 $CH(Q_j)$的顶点,一旦将 p_i 压栈,S 中将恰包含 $CH(Q_j \cup \{p_i\})$的顶点,且依然是从底至顶按逆时针方向的顺序。

现在来说明 $CH(Q_j \cup \{p_i\})$和 $CH(Q_i)$是相同的点集。考虑 **for** 循环的第 i 次重复中任意一个被弹出的点 p_t,并设 p_r 是 p_t 被弹出时栈 S 中恰处于 p_t 下面的点(p_r 可能就是 p_j)。角$\angle p_r p_t p_i$ 构成非左转角,p_t 关于 p_0 的极角大于 p_r 的极角。如图 5-10(b)所示,或在由 p_0、p_r 和 p_i 构成的角的内部,或在该角的边上(但不是该角的顶点)。显然,由于 p_t 在 Q_i 的另外 3 个点构成的角的内部,它不可能是 $CH(Q_i)$的一个顶点,我们有

$$CH(Q_i - \{p_t\}) = CH(Q_i)。\tag{5-1}$$

设 P_i 是 **for** 循环的第 i 次重复期间弹出点构成的集合。由于式(5-1)可用于 P_i 中的所有点,可以重复使用它说明 $CH(Q_i - P_i) = CH(Q_i)$。但是 $Q_i - P_i = Q_j \cup \{p_i\}$,因此推出 $CH(Q_j \cup \{p_i\}) = CH(Q_i - P_i) = CH(Q_i)$。

我们已经说明了一旦把 p_i 压栈,S 中自底向顶恰以逆时针顺序包含 $CH(Q_i)$的各顶点。随着 i 的增长,在下一次重复中循环不变量也成立。

当循环结束时,有 $i = m+1$,所以循环不变量实现了在栈 S 中恰包含 $CH(Q_m)$,也就是 $CH(Q)$ 的各顶点,它们是以逆时针顺序自底向顶置于栈 S 中。这就完成了证明。

图 5-11 所示为 GRAHAM-SCAN 正确性证明。图 5-11(a)中,由于 p_i 关于 p_0 的极角大于 p_j 的极角,且角$\angle p_k p_j p_i$ 左转,将 p_i 添加到 $CH(Q_j)$中将恰得到 $CH(Q_j \cup \{p_i\})$的顶点。图 5-11(b)中,若角$\angle p_r p_t p_i$ 非左转,则 p_t 或在由 p_0、p_r 和 p_i 构成的角的内部,或在该角的边上,总之不可能是 $CH(Q_i)$的顶点。

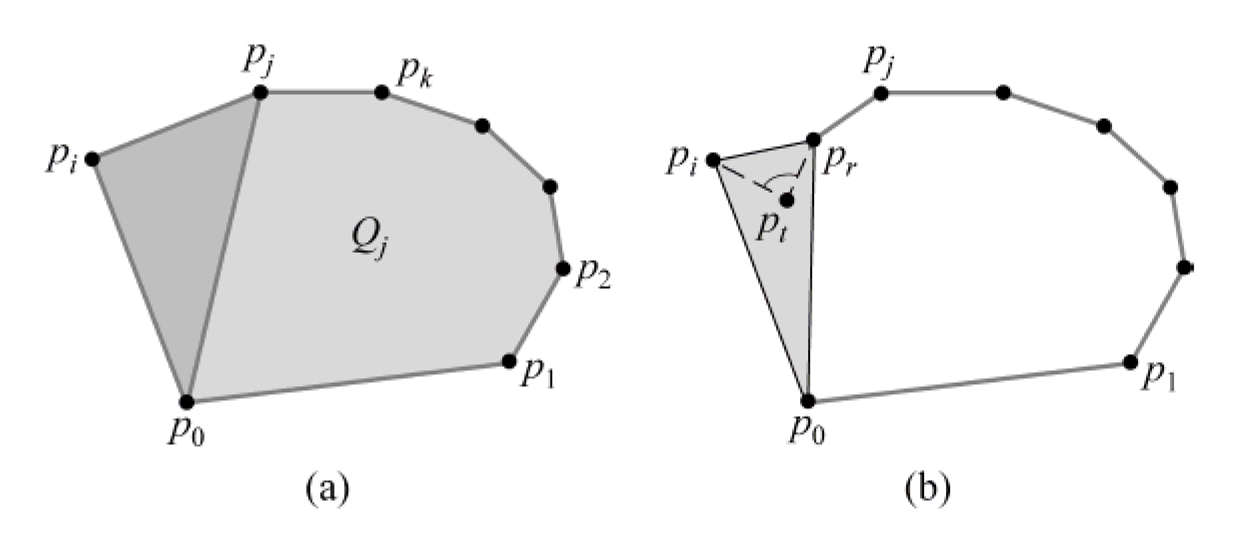


图 5-11 GRAHAM-SCAN 正确性证明

3. 算法分析

现在来说明 GRAHAM-SCAN 的运行时间是 $O(n\lg n)$,其中 $n = |Q|$。第 1 行耗时 $\Theta(n)$。第 2 行利用归并排序或堆排序及动手做算法 5-5 的叉积方法对极角排序耗时 $O(n\lg n)$。第 3～6 行耗时 $O(1)$。因为 $m \leqslant n-1$,第 7～10 行的 **for** 循环至多重复 $n-3$ 次。由于 PUSH 操作耗时 $O(1)$,该循环的每次重复除了第 8～9 行的 **while** 循环,耗时 $O(1)$。因此,该 **for**

循环除了内嵌的 **while** 循环外将耗时 $O(n)$。

下面来说明 **while** 循环全部耗时 $O(n)$。对 $i=0,1,\cdots,m$，每一个点 p_i 仅入栈一次。对每一次 PUSH 都至多有一次 POP 操作。至少有 p_0、p_1 和 p_m 始终不出栈，故总共至多有 $m-2$ 个 POP 操作的执行。**while** 循环的每一次重复执行一个 POP 操作，因此，该 **while** 循环总共至多重复 $m-2$ 次。因为第 8 行的测试耗时 $O(1)$，每次调用 POP 耗时 $O(1)$，并且 $m \leqslant n-1$，此 **while** 循环总耗时 $O(n)$。于是，GRAHAM-SCAN 的运行时间为 $O(n\lg n)$。

5.3.2　程序实现

1. 计算点集中的最低点

算法 5-3 的第一步是在平面点集 Q 中计算纵坐标最小的点作为计算其他各点的极角的基准点 p_0。计算一个全序集（很多情形下是存放于数组中）的最大（小）元素是很多算法中的基本操作，下列程序是一个能对各种类型的数组计算最大（小）元素的函数。

```
1  int most(void *a,int n,int size,int(*comp)(void *,void *)){
2      void *m=malloc(size);
3      int i,k=0;
4      memcpy(m,a,size);/* m ← a[0] */
5      for(i=1;i<n;i++)
6          if(comp(((char*)a+i*size),m)>0){                /* a[i]与 m 比较 */
7              memcpy(m,((char*)a+i*size),size);           /* m ← a[i] */
8              k=i;
9          }
10     free(m);
11     return k;
12 }
```

程序 5-7　计算组成数组的全序集中的最大(小)元素的函数

对程序 5-7 的说明如下。

(1) 函数 most 计算具有 n 个元素（元素的存储宽度为 size）的数组 a 中，按大小比较规则 comp 计算最大（小）元素的下标并作为函数值返回。为了能表示各种不同类型的数组，参数 a 的类型为 **void** *。为适应各种全序关系，元素大小比较规则通过传递给它的函数指针参数 comp 确定。

(2) 函数中，用指针 m 指引的动态存储空间跟踪数组 a 中的最大（小）值，用 k 跟踪最大（小）值的下标，变量 i 用来控制 **for** 循环并表示扫描数组时元素的下标。

(3) 第 4 行调用库函数 memcpy 将 a[0]的值赋于 m 指向的内存空间（第 2 行分配给 m 的）。该函数的使用说明请参见本书 2.1.3 节的相关内容。第 5～9 行的 **for** 循环扫描数组 a，若 a[i]大（小）于 m，则 m 跟踪 a[i]，k 跟踪 i。需要注意的是由于 a 中元素的存储宽度为 i，所以 a+i*size 指向 a[i]。扫描结束时，k 表示最大（小）元素的下标，第 11 行将其返回。在此之前，第 10 行释放 m 指向的内存空间，以防泄漏。

函数 most 是计算表示全序集的数组的最大（小）元素下标的通用函数。为便于代码重

用,将其原型声明保存为文件夹 datastructure 中的头文件 most. h,程序 5-7 则保存为同一文件夹内的源文件 most. c。

在本问题中,需要按照点的纵坐标计算点集的最小者,要利用 most 完成这一操作,需要为参数 comp 定义一个函数。

```
1 static int pyxcomp(Point *a,Point *b){
2     if(a->y<b->y)
3         return 1;
4     if(a->y==b->y&&a->x<b->x)
5         return 1;
6     if(a->y==b->y&&a->x==b->x)
7         return 0;
8     return-1;
9 }
```

程序 5-8　平面点纵坐标的比较

函数 pyxcomp 对参数 a、b 指引的平面点首先比较两者的纵坐标,小则优先(第 2 行和第 3 行)返回 1。若纵坐标相等则比较横坐标,小则优先(第 4 行和第 5 行)返回 1;若两点相同,则返回 0;其余情况返回−1。

若 n 个点构成的点集存放于数组 p,调用函数 most(p,n,**sizeof**(Point),pyxcom)将返回 p 中最低点(纵坐标最小者)的下标。

2. 平面点按极角排序

在本扫描线算法中,事件点进度表是按点集中各点关于用上述的 most 函数找到的纵坐标最小的点 p_0 的极角升序排列的有序线性表。可以调用第 3 章中开发的 quikSort 函数对数组 p 按各点关于 p_0 的极角升序排序。这需要定义平面点 a、b 关于 p_0 的极角大小比较的函数。

```
1 static Point *O;
2 static int angleComp(Point *a,Point *b){          /*比较点 a、b 关于点 O 的极角大小*/
3     double anglea=psudoPolarAngle(a,O),angleb=psudoPolarAngle(b,O);
4     if(anglea==angleb)
5         return 0;
6     if(anglea<angleb)
7         return-1;
8     return 1;
9 }
```

程序 5-9　按极角比较两点大小的函数

第 2～9 行定义的函数 angleComp 比较指针参数 a、b 指向的点关于点 O 的极角的大小。若 a 关于 O 的极角大于 b 关于 O 的极角,函数返回 1;若前者小于后者,函数返回−1;若两者相等,函数返回 0。注意,第 3 行调用程序 5-3 中定义的计算向量伪极角的函数 psudoPolarAngle,分别计算 a 关于 O 的极角和 b 关于 O 的极角。由于该函数作为 quikSort

的参数对事件点进度表 p 排序，它应为函数类型（**int** *）（void *，void *），所以相关点 O（p 中纵坐标最小的点）不能作为函数的参数，因此在第 1 行将其定义为全局变量。

3. 计算平面点集的凸壳

利用程序 5-7 至程序 5-9 中定义的函数，可实现算法 5-3。

```
1  Point *grahamScan(Point* p,int n,int *m){
2      Point *q;
3      Stack *S=createStack(sizeof(Point));
4      int i=most(p,n,sizeof(Point),pyxcomp);
5      swap(p,p+i,sizeof(Point));
6      O=p;
7      quickSort(p+1,sizeof(Point),0,n-1,angleComp);
8      push(S,p);push(S,p+1);push(S,p+2);
9      for(i=3;i<n;i++){
10         ListNode *b=S->top, *a=b->next;
11         while(direction(a->key,b->key,p+i)>0){
12             pop(S);
13             b=S->top;
14             a=b->next;
15         }
16         push(S,p+i);
17     }
18     *m=S->L->n;q=(Point*)calloc(*m,sizeof(Point));
19     for(i=*m-1;i>=0;i--){
20         ListNode *x=pop(S);
21         q[i]=*((Point*)(x->key));
22         clrListNode(x,NULL);
23         free(x);
24     }
25     clrStack(S,NULL);
26     free(S);
27         return q;
28 }
```

程序 5-10　实现算法 5-3 GRAHAM-SCAN 过程的 C 函数

对程序 5-10 的说明如下。

（1）用平面点 Point 型数组 p 表示平面点集 Q，并为参数。参数 n 表示 Q 中点的个数。与算法 5-3 稍有不同的是，为便于运用，函数 grahamScan 并不是直接返回存放凸包的栈，而是将栈中的点转存为一个数组，然后返回该数组。由于 C 语言的数组不含元素的个数属性，所以用指针参数 m 来指引所返回数组的元素个数。

（2）第 3 行用第 2 章程序 2-12 定义的类型 Stack 声明栈 S。第 4 行调用程序 5-7 中的函数 most 计算列表 p 中纵坐标最小者的下标 i。注意，调用 most 时传递给它的第 4 个参数是指向程序 5-8 定义的函数 pyxcomp 的指针。第 5 行调用第 3 章程序 3-3 定义的函数

swap,交换数组 p 中的 p[0]、p[i]使得 p[0]为 p 中的纵坐标最小者。第 6 行使全局变量 O 指向 p[0]。第 7 行调用第 3 章程序 3-10 定义的函数 quikSort,对 p[1..n－1]按各点关于 O 的极角升序排序。该函数的原型是

```
void quickSort(void * a,int size,long p,long r,int( * comp)(void * ,void * ));
```

其中,参数 a 表示需排序的数组,参数 size 表示数组元素的存储宽度,参数 p、r 表示子数组的起点和终点下标,而第 5 个参数 comp 表示存储于 a 中的全序集的元素大小比较规则。

注意传递给 quikSort 的第 5 个参数是程序 5-9 中定义的函数 angleComp 的指针。这样就完成了算法第 1 行和第 2 行的操作。

(3) 第 8 行,将 p[0]、p[1]和 p[2]一次压入栈 S,实现算法中的第 3～5 行操作。第 9～17 行的 **for** 循环对应算法中第 6～9 行的 **for** 循环。其中的 a 对应 NEXT-TO-TOP(S),b 对应 TOP(S),第 11～15 行的 **while** 循环完成算法中第 7 行和第 8 行的 **while** 循环。第 12 行和第 16 行调用第 2 章程序 2-13 中定义的函数 pop 和 push 完成对 S 的弹出和压栈操作。第 18 行为存放要返回的凸包的数组 q 分配空间,第 19～24 行的 **for** 循环一次将栈 S 中的点存放到 q 中。注意存放的顺序是将栈的数据反向存入 q,这样 q 中的点就是凸包的按逆时针方向的序列了。

为便于重用,程序 5-7 和程序 5-8 的代码保存在文件夹 geometry 中的源文件 convexhull. c 中。函数 grahamScan 的原型声明存储在同一文件夹中的头文件 convexhull. h 中。

5.4 求最邻近点对

现在来考虑求 $n\geqslant 2$ 个点的集合 Q 中最邻近点对的问题。“最邻近”指的是欧几里德距离:两点 $p_1=(x_1,y_1)$和 $p_2=(x_2,y_2)$之间的距离是 $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$。集合 Q 的两个点可能重合,此时,它们之间的距离为零。这个问题在交通控制中有其应用。控制航空或航海交通的系统为了检测载体之间潜在的碰撞,可能需要知道哪两个载体最邻近。

把问题形式化为如下。

输入:平面点集 Q。

输出:Q 中最邻近点对的距离 δ。

计算最邻近点对的蛮力算法直接查看 $\binom{n}{2}=\Theta(n^2)$个点对。在本节中,将描述一个对此问题的一个分治算法,其运行时间用我们熟悉的递归式 $T(n)=2T(n/2)+O(n)$描述。于是,此算法仅用 $O(n\lg n)$时间。

5.4.1 算法描述与分析

1. 分治算法

算法的每一次递归调用以子集 $P\subseteq Q$ 和包含 P 中点的数组 X 及 Y 作为输入。数组 X 中的点按横坐标的单调递增排序。类似地,数组 Y 按点的纵坐标单调增加排序。为了达到

$O(n\lg n)$的时间界限，不能在每次递归调用时排序。如果这样，递归的运行时间将变成 $T(n)=2T(n/2)+O(n\lg n)$，其解为 $T(n)=O(n\lg 2n)$。稍后将看到如何使用“预排序”来维持这一顺序特性而无须在每次递归调用时排序。

给定输入为 P、X 和 Y 的一次递归调用，首先检测是否$|P|\leqslant 3$。如是，则该次调用直接执行上述的蛮力法：试探所有$\binom{|P|}{2}$个点对并返回最邻近点对。若$|P|>3$，递归调用执行如下的分治范例。

分割：找一竖线 l，将集合 P 分成两半 P_L 和 P_R，使得$|P_L|=\lceil |P|/2\rceil$，$|P_R|=\lfloor |P|/2\rfloor$，$P_L$中的所有点都位于直线 l 上或其左边，而 P_R中的所有点都位于直线 l 上或其右边。数组 X 被分成两部分 X_L和 X_R，分别包含 P_L和 P_R的点，并按横坐标的单调增加排序。类似地，数组 Y 被分成两部分 Y_L和 Y_R，分别包含 P_L和 P_R的点，并按 y 坐标的单调增加排序。

治理：把 P 分割成 P_L和 P_R，就形成两个递归调用，一个是求 P_L的最邻近点对，另一个是求 P_R的最邻近点对。第一个调用的输入是 P_L、X_L和 Y_L，第二个调用的输入是 P_R、X_R和 Y_R。设 P_L、P_R的最邻近点对的距离分别为 δ_L和 δ_R，并设 $\delta=\min(\delta_L,\delta_R)$。

合并：最邻近点对或是由某个递归调用找到的距离为 δ 的点对，或是一个点在 P_L中另一个点在 P_R中的点对。算法判断是否存在这样的距离小于 δ 的点对。显然，如果有这样的距离小于 δ 的点对，如图 5-12(a)所示，这样的点必落在对称于直线 l、宽 2δ 的竖直条状区域内。为找到这样的点对，算法做如下工作。

(1) 创建数组 Y'，它是将数组 Y 中所有不在 2δ 宽的条形区域内的点去掉的结果。数组 Y'与 Y 一样按纵坐标排序。

(2) 对 Y'中的每个点 p，算法试图在 Y'中找与 p 的距离小于 δ 的点。在 Y'中只需要考虑 p 之后的 7 个点。算法计算从 p 到这 7 个点的距离并跟踪在 Y'中找到的最邻近点对的距离 δ'。

(3) 若 $\delta'<\delta$，则条状区域内确实包含比递归调用所得到的更邻近的点对。该点对及其距离 δ'被返回，否则返回由递归调用得到最邻近点对及其距离 δ。

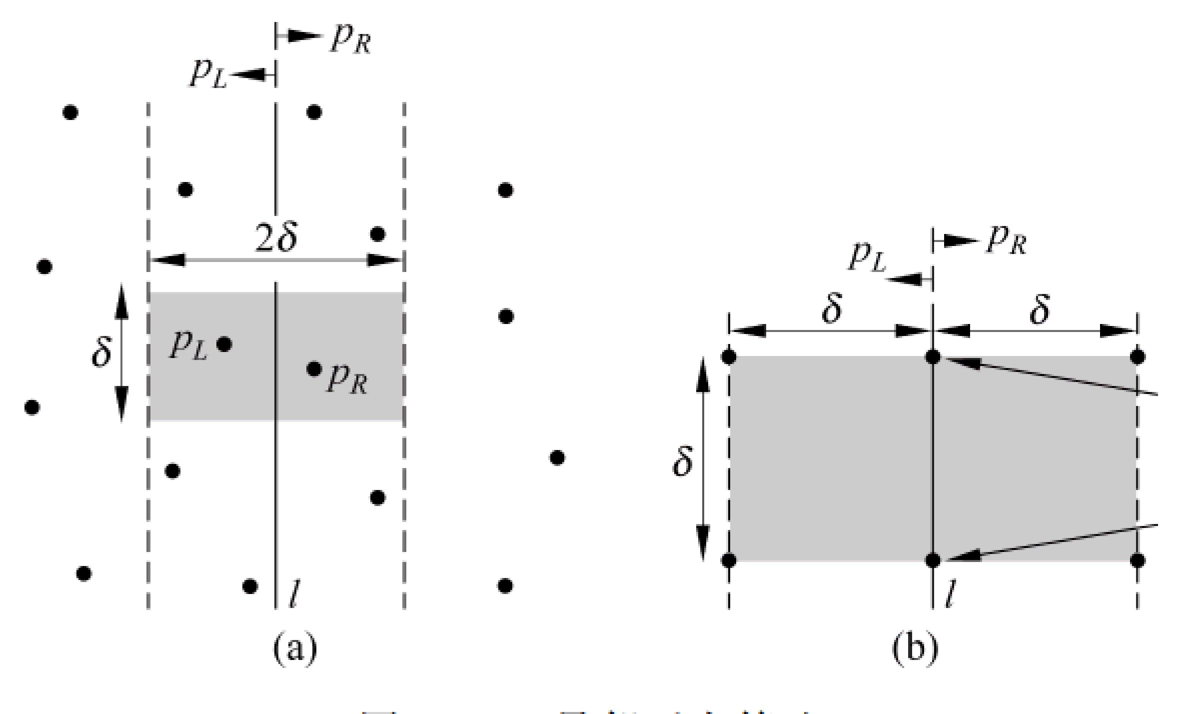


图 5-12　最邻近点算法

图 5-12 所示为证明最邻近点对算法对数组 Y'中的每个点只需检测其后的 7 个点的关键概念。图 5-12(a)中，若 $p_L\in P_L$，$p_R\in P_R$的距离小于 δ，则必居于关于直线 l 对称的 $\delta\times 2\delta$ 的矩形中。图 5-12(b)中，在 $\delta\times\delta$ 方块内相距至少 δ 单位的 4 个点。左边是 P_L中的 4 个

点,右边是 P_R 中的 4 个点,共有 8 个点在 $\delta\times2\delta$ 矩形内。落在直线 l 上的两对是重合的。

上面的描述省略了一些为获得 $O(n\lg n)$ 运行时间的实现细节。在证明了算法的正确性后,将说明如何实现期望的时间界限。

```
CLOSEST-PAIR-POINTS(P,X,Y)
if |P|<3
      then 蛮力计算所有点对的距离并返回最小者
计算竖直线 l 将 P 分成两部分 P_L 和 P_R,使得|P_L|=⌈|P|/2⌉,|P_R|=⌊|P|/2⌋
X_L ← P_L 中所有点,X_R ← P_R 中所有点,并按横坐标的升序排好序
Y_L ←P_L 中所有点,Y_R ←P_R 中所有点,并按纵坐标的升序排好序
δ_L ←CLOSEST-PAIR-POINTS(P_L ,X_L ,Y_L)
δ_R ←CLOSEST-PAIR-POINTS(P_R ,X_R ,Y_R)
δ_m ←min(δ_L ,δ_R)
d←COMBINE(P,l,δ_m)
δ← min(δ_m ,d)
return δ
COMBINE (P,l,δ)
Y'←P 中所有位于以 l 为中轴线,宽度为 2δ 的带状区域内的点
对 Y'中的点按纵坐标升序排序
for Y'中的每一个点 p
      do 计算 Y'中所有距离不超过 δ_m 的顶点距离,并跟踪最小者为 d
return d
```

算法 5-6　解决最邻近点对问题的算法

2. 正确性

为说明这个最邻近点对算法的正确性,需要考察以下两个问题。首先,当 $|P|\leqslant3$ 时递归到底,保证不会遇到划分的点集仅有一个点;其次,只需要检测 Y' 中的所有 7 个点。现在就来证明这一特性。

假定在某一层递归中,最邻近点对为 $p_L\in P_L$,$p_R\in P_R$。于是 p_L 和 p_R 之间的距离 δ' 必严格小于 δ。点 p_L 必落于直线 l 的左边或 l 上,并且距离小于 δ 个单位。同样,点 p_R 必落于直线 l 的右边或 l 上,并且距离小于 δ 个单位。此外,p_L 和 p_R 在竖直方向上的距离小于 δ。于是,如图 5-12(a)所示,p_L 和 p_R 落于对称于 l 的 $\delta\times2\delta$ 的矩形区域内(此矩形区域内可能还有其他的点)。

下面来证明在 $\delta\times2\delta$ 的矩形内至多有 8 个 P 的点。考虑该矩形右边的 $\delta\times\delta$ 的方块。由于 P_L 中的点至少相距 δ 个单位,因此至多有 4 个点落在此方块中,如图 5-12(b)所示。相似地,至多有 4 个 P_R 中的点落在右半边的 $\delta\times\delta$ 的方块中。于是,至多 8 个 P 中的点居于此 $\delta\times2\delta$ 的矩形内(由于 l 上的点可能既是 P_L 的也是 P_R 的,因此 l 上的点可多达 4 个。当两对点重合时,达到此极限,每一对点既是 P_L 的也是 P_R 的,一对为矩形与 l 在顶部的交点,另一对为矩形与 l 在底部的交点)。

说明了至多有 8 个 P 中的点能居于此矩形内,就不难看出对 Y' 中的每一个点,只需检测其后的 7 个点。假定最邻近点对为 p_L 和 p_R,不失一般性,设在 Y' 中 p_L 先于 p_R,则无论 p_L 先于 p_R 多远,p_R 是要检测的 7 个点之一。这就说明了最邻近点对算法的正确性。

3. 实现与运行时间

正如已经指出的那样，我们的目标是要使递归的运行时间是 $T(n)=2T(n/2)+O(n)$。其中，$T(n)$为对 n 个点的集合的运行时间。主要的困难是保证数组传递给递归的 X_L、X_R、Y_L和 Y_R是已按合适的坐标排序的并且 Y'是按纵坐标排好序的(注意若递归接受的数组 X 已排序，则将 P 划分成 P_L和 P_R很容易在线性时间内完成)。

关键是在每次调用希望对一个有序数组形成一个有序的子集。例如，一次调用给定子集 P 和按 y 坐标排好序的数组 Y。在把 P 划分成 P_L和 P_R的同时，要形成按 y 坐标排好序的数组 Y_L和 Y_R。此外，这些数组必须在线性时间内形成。这一方法可以视为 2.3.1 节的合并排序过程 MERGE 的反向操作：要把一个已排序的数组分裂成两个排好序的数组。以下代码给出了这一想法。

```
length[Y_L] ← length[Y_R] ←0
for i←1 to length[Y]
    do if Y[i]∈P_L
        then length[Y_L] ←length[Y_L]+1
            Y_L[length[Y_L]] ←Y[i]
        else length[Y_R] ←length[Y_R]+1
            Y_R[length[Y_R]] ←Y[i]
```

算法 5-7　将有序列表 Y 分解成两个有序子列表 Y_L和 Y_R

依次检测数组 Y 中的点，如果点 $Y[i]$在 P_L中，将其追加到数组 Y_L的尾部，否则追加到 Y_R的尾部。对数组 X_L、X_R和 Y'的代码是相似的。

所剩的最后问题是如何在一开始就使所有的点都排好序。直接对其**预排序**来解决这一问题，即在第一次递归调用前就把它们一次性全部排好序。这些有序数组传递给第一个递归调用，由此，在进一步递归调用需要的时候对其进行分割。预排序增加运行时间 $O(n\lg n)$，但在每一步递归时只需花费线性时间。于是，设 $T(n)$为每一步递归时间，$T'(n)$为整个算法的时间，有 $T'(n)=T(n)+O(n\lg n)$ 和 $T(n)=\begin{cases}2T(n/2)+O(n) & n>3\\ O(1) & n\leqslant 3\end{cases}$ 于是，$T(n)=O(n\lg n)$且 $T'(n)=O(n\lg n)$。

5.4.2　程序实现

1. 蛮力算法的实现

当点集 P 中的个数不超过 3 时，用下列函数计算其中最邻近点对的距离。

```
double force(Point * p,int n){          /* 计算最邻近点对距离的蛮力算法实现函数 */
    double d=DBL_MAX,t;
    int i,j;
    for(i=0;i<n;i++)
        for(j=i+1;j<n;j++){
            t=dist(&p[i],&p[j]);
```

```
7                if(t<d)
8                    d=t;
9            }
10       return d;
11 }
```

程序 5-11　蛮力计算点集最邻近点对距离的函数

该函数的第一个参数 p 是表示平面点集的数组 p，第二个参数 n 表示 p 中所含点的个数。函数返回 p 中所有点对距离的最小者。

函数体中定义了用来跟踪最邻近点对距离的变量 d。初始时，d 的值应为∞，用头文件 limits.h 中定义的系统常量 DBL_MAX 对其进行初始化。

第 4～9 行的两重嵌套 **for** 循环计算 p 中所有 p[i]、p[j]($i \neq j$)之间的距离 t，只要 t<d，就将 d 改写为 t。这样，循环结束时，d 记录了其中的最小者。第 10 行将 d 返回。

2. 合并函数

分治算法 CLOSEST-PAIR-POINTS 在将 P 划分为 P_L 和 P_R，并对 P_L 和 P_R 分别调用自身分治后，还用调用合并过程 COMBINE，得到对 P 的解。过程 COMBINE 实现如下。

```
double combine(Point *y,int n,double lx,double delta){
    int i,j;
    double d=DBL_MAX,t;
    Point* y1=(Point*)malloc(n*sizeof(Point));
    for(i=0,j=0;i<n;i++)          /*将 y 中宽度为 2*delta 的带状区域内的子集置于 y1*/
        if(fabs(y[i].x-lx)<=delta)
            y1[j++]=y[i];
    n=j;                          /*y1 中的元素个数*/
    for(i=0;i<n;i++){             /*计算 y1 中最邻近点对距离*/
        for( j=1;i+j<n&&j<8;j++){
            t=dist(&y1[i],&y1[i+j]);
            if(t<d)
                d=t;
        }
    }
    free(y1);
    return d;
}
```

程序 5-12　实现算法 5-4 中 COMBINE 过程的 C 函数

对程序 5-12 的说明如下。

(1) 函数 combine 的参数 y 表示点集中点按纵坐标升序排列的数组，参数 n 表示 y 中元素个数。参数 lx 表示对点集划分的中轴线，参数 delta 表示对点集的两个划分子集递归解得的子问题解的较小者。该函数返回点集中以 lx 为中轴线，宽度为 2*delta 的带状区域内最邻近点对距离。

（2）函数体中，第 4 行定义了指向点集中位于上述带状区域内点组成的数组的指针 y1。

（3）第 5～7 行的 for 循环将 y 中位于带状区域内的元素依次复制到 y1 中。这样，可保证 y1 中的点也是按纵坐标升序排列。第 9～15 行对 y1 中的每一个点计算与其邻近的 7 个点的距离，跟踪最小者 d。

3. 计算最邻近点对距离

利用程序 5-11 和程序 5-12，实现算法 5-6 计算平面点集中最邻近点对距离的函数定义如下。

```
double closestPairPoints(Point *x,Point *y,int n){
    int half,k,i,j;
    double lx,deltal,deltar,delta,d;
    Point *xl,*xr,*yl,*yr;
    if(n<=3)                /*若点数不超过 3,蛮力计算所有点对的距离并返回最小者*/
        return force(x,n);
    half=n/2;
    xl=(Point*)malloc(half*sizeof(Point)),
    xr=(Point*)malloc((n-half)*sizeof(Point)),
    yl=(Point*)malloc(half*sizeof(Point)),
    yr=(Point*)malloc((n-half)*sizeof(Point));
    assert(xl&&xr&&yl&&yr);
    memcpy(xl,x,half);                /*将 x 按点的横坐标升序分解成两个集合*/
    memcpy(xr,x+half,n-half);
    lx=x[half].x;
    for(k=i=j=0;k<n;k++)              /*将 y 分解成对应的两个按纵坐标升序集合*/
        if(y[k].x<=lx&&i<=half)
            yl[i++]=y[k];
        else
            yr[j++]=y[k];
    deltal=closestPairPoints(xl,yl,half);       /*对左半部分递归*/
    deltar=closestPairPoints(xr,yr,n-half);     /*对右半部分递归*/
    delta=deltal<deltar?deltal:deltar;
    d=combine(y,n,lx,delta);
    free(xl);free(yl);free(xr);free(yr);
    return d<delta?d:delta;
}
```

程序 5-13　计算平面点集中最邻近点对距离的 C 函数

对程序 5-13 的说明如下。

（1）CLOSEST-PAIR-POINTS 过程一共有 3 个参数：平面点集 P、P 中元素按横坐标升序排列后的序列 X 以及 P 中元素按纵坐标升序排列后的序列 Y。作为实现算法的程序，函数 closestPairPoints 只需要传递排好序的序列 X 和 Y 就可以了，这两个参数都用元素类型为 Point 的数组 x、y 表示。同时需要参数 n 说明数组 x、y 的元素个数。CLOSEST-PAIR-POINTS 过程返回点集中最邻近点对的距离 δ，应该是浮点型的。所以函数 closestPairPoints 的返回值类型为 double 型的。

(2) 在 CLOSEST-PAIR-POINTS 过程中要对 X 和 Y 分解，分解后的子向量为 X_L、X_R、Y_L、Y_R。同时还需要设置下一层递归返回值 δ_L、δ_R，两者的最小值 δ_m，以及合并过程 COMBINE 的返回值 d。在程序中，用指针 xl、xr、yl、yr 指向的动态数组表示 X_L、X_R、Y_L、Y_R。用 deltal、deltar、delta 表示 δ_L、δ_R、δ_m。COMBINE 的返回值 d 用同名的 d 表示。

(3) 函数体中第 5 行和第 6 行对 p 中点数 n 不超过 3 的情形调用程序 5-11 定义的函数 force 强力计算最邻近点对距离。第 13 行和第 14 行完成对 x 的划分，将前一半 x[0..n/2－1] 复制给 xl，x[n/2..n]复制给 xr。第 15～20 行是按算法 5-6 实现的对 y 的划分。第 21 行和第 22 行分别对 xl、yl 及 xr、yr 进行递归操作，返回 deltal 和 deltar。第 23 行计算 deltal 和 deltar 的最小者，第 24 行调用程序 5-12 中定义的函数 combine 计算 y 中以 lx 为中轴，宽度为 2 * delta 的区域内最邻近点对距离。第 26 行返回 y 中最邻近点对距离。

程序 5-11 至程序 5-13 定义的函数存储在文件夹 geometry 中的源文件 closestpairpoints.c 中。函数 closestPairPoints 的原型声明存储在同一文件夹中的头文件 closestpairpoints.h 中。

5.5 应用

5.5.1 光导管

PiPe

The GX Light Pipeline Company started to prepare bent pipes for the new transgalactic light pipeline. During the design phase of the new pipe shape the company ran into the problem of determining how far the light can reach inside each component of the pipe. Note that the material which the pipe is made from is not transparent and not light reflecting.

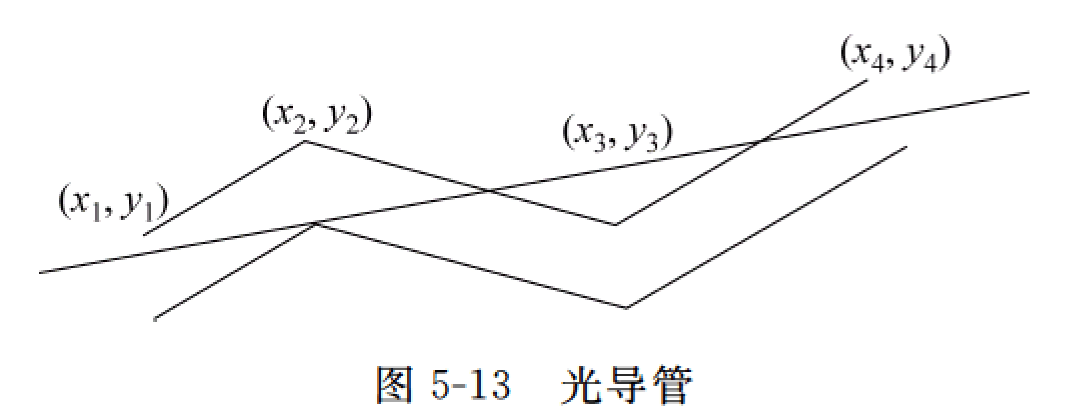


图 5-13 光导管

Each pipe component consists of many straight pipes connected tightly together. For the programming purposes, the company developed the description of each component as a sequence of points $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$, where $x_1 < x_2 < \cdots < x_n$. These are the upper points of the pipe contour. The bottom points of the pipe contour consist of points with y-coordinate decreased by 1. To each upper point (x_i, y_i) there is a corresponding bottom point $(x_i, y_i - 1)$ (see picture above). The company wants to find, for each pipe component, the point with maximal x-coordinate that the light will reach. The light is

emitted by a segment source with endpoints (x_1, y_1-1) and (x_1, y_1) (endpoints are emitting light too). Assume that the light is not bent at the pipe bent points and the bent points do not stop the light beam.

Input

The input file contains several blocks each describing one pipe component. Each block starts with the number of bent points $2 \leqslant n \leqslant 20$ on separate line. Each of the next n lines contains a pair of real values x_i, y_i separated by space. The last block is denoted with $n=0$.

Output

The output file contains lines corresponding to blocks in input file. To each block in the input file there is one line in the output file. Each such line contains either a real value, written with precision of two decimal places, or the message through all the pipe. The real value is the desired maximal x-coordinate of the point where the light can reach from the source for corresponding pipe component. If this value equals to x_n, then the message through all the pipe. will appear in the output file.

Sample Input

```
4
0 1
2 2
4 1
6 4
6
0 1
2 -0.6
5 -4.45
7 -5.57
12 -10.8
17 -16.55
0
```

Sample Output

```
4.67
Through all the pipe.
```

1. 问题描述与分析

光缆公司设计一条光导管，它由若干条光线不能穿透管壁、壁管材料也不能反射光线的直管段连接而成（如图 5-13 所示）。连接处的顶部坐标为(x_i, y_i)，底部坐标为(x_i, y_i-1)，$i=1,2,\cdots,n$，$x_1<x_2<\cdots<x_n$。光线从直径为 1 的入口处射入。对给定接口顶部坐标序列(x_1, y_1)，(x_2, y_2)，…，(x_n, y_n)（隐含地给出了接口底部的坐标(x_1, y_1-1)，(x_2, y_2-1)，…，(x_n, y_n-1)）的管道设计方案，问从入口射入的光线最多能传送多远？问题的形式化表述为如下。

输入：管道连接处顶部坐标序列$(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)$，$x_1< x_2<\cdots< x_n$(隐含地给出了接口底部的坐标$(x_1,y_1-1),(x_2,y_2-1),\cdots,(x_n,y_n-1)$)。

输出：若光线能穿透整个管道则输出“穿透”信息；否则，输出从入口射入光线能在管道中传送的最远距离。

此问题中，设光线穿透第 i 根管子的角度(光线与水平线的夹角)范围为$[low_i,up_i]$(见图 5-14)，对于光线能穿透整个管道的条件比较简单：$\bigcap_{i=1}^{n}[low_i,up_i]\neq\varnothing$。

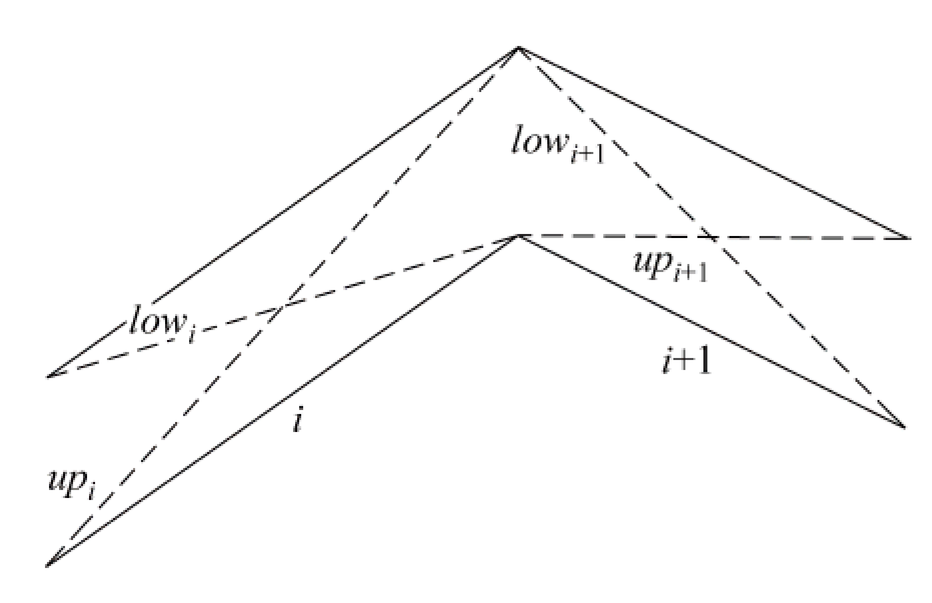


图 5-14　光线通透管子的角度

对光线不能穿透整条管道的情形，先给出如下结论。

(1) 最长光线至少擦过一个连接口的顶部或底部。

(2) 最长光线必擦过一个接口的顶部，另一个接口的底部。

对于结论(1)，如果最长光线不擦过任何一个接口处的顶部或底部，则经过适当的调整射入光线角度，就可得到一条从入口射入，擦过某个接口处的顶部或底部，且比原来的光线更长的光线。

对于结论(2)，不失一般性，假定最长光线仅擦过若干个接口顶部，也可以通过适当调整射入光线角度，而得到一条从入口射入，在管道内擦过某个接口的顶部，且擦过某个接口的底部的更长的光线。

2. 算法描述

根据上述两个结论，可以先计算出光线通过每一段管子的角度范围 low_i 和 up_i，并以此算出从入口开始能在管道中穿过的最长距离。

```
PIPE(x,y)
intersection ←(−∞,+∞)
k ← 1
repeat
   low_k←第 k 段管子的最小入射角
   up_k←第 k 段管子的最大入射角
   intersection ← intersection ∩ [low_k,up_k]
   k ← k+1
until k>n or intersection=∅
if intersection≠∅
   then return ∞
k ← 1,len ← 0
```

```
12 intersection ← intersection－[low_k ,up_k]
13 for each 管路中的拐角点对 a、b
14   do α ← a、b 连线的极角
15     if α∈ intersection
16       then temp ← a、b 连线从入射点到在管路中第 1 次碰壁处长度
17         if temp＞len
18           then len ← temp
19 return len
```

算法 5-8　解决 PiPe 问题的算法过程

算法运行如下。第 3～8 行的 **repeat-until** 循环依次计算每一根管道的最大/最小光线入射角(光线与横轴正向夹角)low_i 和 up_i，并计算 $\bigcap_{i=1}^{k}[low_i, up_i](k \leqslant n)$。第 9 行检测条件 $\bigcap_{i=1}^{n}[low_i, up_i] \neq \varnothing$ 是否成立。若是，意味着光线可穿透整个管路，返回∞。否则，第 12 行在交集 $\bigcap_{i=1}^{k}[low_i, up_i] = \varnothing$ 中减去最后一次交上去的区间$[low_k, up_k]$，这样必有 $\bigcap_{i=1}^{k-1}[low_i, up_i] \neq \varnothing$。第 12～17 行的 **for** 循环逐一对管路中拐角点对的连线检测与横轴正向夹角是否在 $\bigcap_{i=1}^{k-1}[low_i, up_i]$ 范围内。若是，计算该连线在管路中与管子碰壁处距射入点的长度。跟踪最大者 len。循环结束时，第 19 行返回 len。

3. 程序实现

1) 计算光线的入射角

与计算向量的极角有所不同，光线 l 的入射角 α 定义为光线与 x 轴正向的夹角，即 $-\pi \leqslant \alpha \leqslant \pi$。计算过点 a、b 的直线与 x 轴夹角近似值的过程实现如下。

```
1 double psudoAngle(Point * a,Point * b){
2   Point p1=sub(a,b);
3   assert(pabs(&p1)>=epsilon);
4   normalize(&p1);
5   return p1.y;
6 }
```

程序 5-14　计算点 a、b 连线与 x 轴正向夹角近似值的 C 函数

函数中第 2 行计算连接点 a、b 的向量 p1。第 4 行将 p1 规格化(使其模长为 1)，第 5 行用 p1 的纵坐标作为 p1 与 x 轴正向夹角的近似值返回。

2) 计算线段交点

线段所在直线对应一个二元一次方程。两条相交线段可表为一个二元一次方程组 $\begin{cases} a_{11}x+a_{12}y=b_1 \\ a_{21}x+a_{22}y=b_2 \end{cases}$。令解出该方程组 $d_x=\begin{vmatrix} a_{11} & b_1 \\ a_{21} & b_2 \end{vmatrix}$，$d_y=\begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \end{vmatrix}$ 及 $d=\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$，解该方程 $x_0=d_x/d, y_0=d_y/d, (x_0, y_0)$即为相交线段的交点坐标。程序 5-15 就是用此方法计算相交线段交点的。

```
Point jointPoint(Point * p1,Point * p2,Point * p3,Point * p4){
```

```
2    double a11,a12,a21,a22,b1,b2,d,dx,dy;
3    Point p;
4    a11=(p2->y-p1->y);a12=(p1->x-p2->x);b1=a11*(p2->x)+a12*(p2->y);
5    a21=(p4->y-p3->y);a22=(p3->x-p4->x);b2=a21*(p4->x)+a22*(p4->y);
6    d=a11*a22-a21*a12;
7    dx=b1*a22-b2*a12;
8    dy=a11*b2-a21*b1;
9    p.x=dx/d;p.y=dy/d;
10   return p;
11 }
```

程序 5-15　计算相交线段 p1、p2 与线段 p3、p4 交点的 C 函数

程序中第 4 行和第 5 行根据点 p1、p2、p3、p4 的坐标计算方程组系数 a11、a12、b1、a21、a22、b2。第 6~8 行计算 d、dx 和 dy。第 9 行计算交点 p 的横坐标 dx/d、纵坐标 dy/d,第 9 行返回 p。

为便于代码重用,将程序 5-15 存在 geometry 文件夹中的头文件 point.h(函数原型声明)及源文件 point.c(函数定义代码)中。

3) **计算光线在管路中的长度**

接下来实现过管路中两个拐角顶点的光线的长度的计算函数。

```
double calculate(double *x,double *y,int i,int j,int iup,int jup,int n){
    int k;
    Point a,b,left,right,p;                          /*计算过点 a、b 的线段 left、right*/
    a.x=x[i];b.x=x[j];                               /*还原 a、b*/
    a.y=iup?y[i]:y[i]-1;
    b.y=jup?y[j]:y[j]-1;
    left.x=x[0];right.x=x[n-1];                      /*左、右端点的横坐标*/
    left.y=(b.y-a.y)/(b.x-a.x)*(left.x-a.x)+a.y;     /*左端点的纵坐标*/
    right.y=(b.y-a.y)/(b.x-a.x)*(right.x-a.x)+a.y;   /*右端点的纵坐标*/
    a.x=x[j];a.y=jup?y[j]-1:y[j];                    /*第 j 段管子管壁起点*/
    b.x=x[j+1];b.y=jup?y[j+1]-1:y[j+1];              /*第 j 段管子的终点*/
    if(segmentsIntersect(&a,&b,&left,&right)){       /*若 a、b 和 left、right 相交*/
        p=jointPoint(&a,&b,&right,&left);            /*计算交点*/
        return dist(&left,&p);                       /*计算光束长度*/
    }
    k=j+1;
    while(k<n-1){          /*从第 j+1 段管子起搜索 left、right 相交的管壁*/
        a.x=x[k];a.y=y[k];
        b.x=x[k+1];b.y=y[k+1];
        if(segmentsIntersect(&a,&b,&left,&right)){
            p=intersection(&a,&b,&right,&left);
            return dist(&left,&p);
        }
        a.y--;b.y--;
        if(segmentsIntersect(&a,&b,&left,&right)){
```

```
26              p=intersection(&a,&b,&right,&left);
27              return dist(&left,&p);
28          }
29          k++;
30      }
31 }
```

程序 5-16　计算过管路中两个拐角点的光线在管路中碰壁前的长度

对程序 5-16 的说明如下。

(1) 参数 x 和 y 分别表示各管子上边缘端点的横坐标数组和纵坐标数组。参数 n 表示构成管路的管子数。参数 i 和 j 分别表示前后两个拐角的编号。参数 iup、jup 分别表示前后两个拐角点是否为上边缘的。函数计算返回过由 i、j、iup、jup 决定的两个拐角点的光线在管路中首次碰壁前的长度。

(2) 第 7 行决定光线上两点 left 和 right(left 位于管路入口处,right 位于管路出口处)的横坐标。第 8 行和第 9 行根据两个拐角点 a、b(第 4～6 行根据参数 x、y、i、j、iup、jup 计算而得)计算出光线上 left 处的纵坐标和 right 处的纵坐标(用解析几何中直线的两点式方程计算)。第 10 行和第 11 行计算出第 j 段管子可能与光线相交的关闭端点 a、b。第 12 行通过调用程序 5-2 中定义的函数 segmentsIntersect 检测线段 left、right 与线段 a、b 是否相交。若是,第 13 行调用程序 5-15 定义的函数 jointPoint 计算两者的焦点 p,第 14 行调用程序 5-1 中定义的函数 dist 计算 left、p 之间的距离作为函数值返回。

(3) 若第 12 行的检测不真,及线段 left、right 并不与 a、b 相交,第 17～30 行的 **while** 循环从第 j+1 根管子起逐一检测管壁是否与 left、right 相交,一旦检测到相交的情况,计算出 left 到交点 p 的距离即为所求,并加以返回。

4) **计算最长光线**

做好这些准备工作后,把算法 5-7 实现为如下 C 函数。

```
double pipe(double * x,double * y,int n){
  double angle1,angle2,len=0,l=-DBL_MAX,u=DBL_MAX,low,up;
  int i=1,j;
  Point a,b,c,d;
  do{                    /* 计算通过各个管子光线入射最大角度、最小角度跟踪公共角度范围 */
      double m1,m2;
      low=l;
      up=u;
      a.x=x[i-1],a.y=y[i-1],b.x=x[i],b.y= y[i]-1;
      angle1=psudoAngle(&b,&a);
      a.y=y[i-1]-1;b.y=y[i];
      angle2=psudoAngle(&b,&a);
      m1=(angle1<angle2)?angle1:angle2;
      m2=angle1+angle2-m1;
      if(m1>l)
          l=m1;
      if(m2<u)
```

```
18                u=m2;
19            i++;
20        }while(i<n&&l<=u);
21        if(l<=u)                                /* 穿透 */
22            return DBL_MAX;
23        for(i=0;i<n-1;i++){                     /* 计算过每对合法点光束的长度,跟踪最大者 */
24            a.x=b.x=x[i];a.y=y[i];b.y=y[i]-1;                /* 左边点 */
25            for(j=i+1;j<n;j++){
26                double temp;
27                c.x=d.x=x[j];c.y=y[j];d.y=y[j]-1;            /* 右边点 */
28                angle1=psudoAngle(&a,&d);                    /* 过线段 ad 光线的入射角 */
29                if(angle1>=low&&angle1<=up){
30                    temp=calculate(x,y,i,j,1,0,n);
31                    if(temp>len)
32                        len=temp;
33                }
34                angle1=psudoAngle(&b,&c);                    /* 过线段 bc 光线的入射角 */
35                if(angle1>=low&&angle1<=up){
36                    temp=calculate(x,y,i,j,0,1,n);
37                  if(temp>len)
38                      len=temp;
39                }
40            }
41        }
42        return len;
43    }
```

程序 5-17　实现算法 5-6 的 C 函数

对程序 5-17 的说明如下。

(1) 第 5～20 行的 **do-while** 循环实现算法过程中第 3～8 行的 **repeat-until** 循环,逐一计算各根管子光线的最大入射角、最小入射角,计算它们的交集。交集为一区间,左端点为 l,右端点为 u。当此交集为空时($l>u$)或所有管子的光线射入角的允许范围交集非空,循环结束。若后者发生($l\leqslant u$),意味着光线可从入口穿透整个管路,第 22 行返回 DBL_MAX,代表光线无穷长。

(2) 第 23～41 行的两重 **for** 循环对 $n(n-1)$ 对管子拐角点连线逐一检测其入射角(调用程序 5-14 定义的函数 psudoAngle 计算)是否介于 l、u 之间。若是,调用程序 5-16 定义的 calculate 函数计算着一条光线在管路中的长度,跟踪最大者 len。

5) 主函数

下列主函数最终解决 PiPe 问题。

```
int main(){
    double *x=NULL, *y=NULL, *y1=NULL,length;
    int n,i;
    FILE *f1=fopen("chap05/Pipe/inputdata.txt","r"),
```

```
5          *f2=fopen("chap05/Pipe/outputdata.txt","w");
6      assert(f1&&f2);
7      fscanf(f1,"%d",&n);                          /*读取案例数 n*/
8      while(n>0){
9          if(x)free(x);
10         if(y)free(y);
11         assert(x=(double*)malloc(n*sizeof(double)));
12         assert(y=(double*)malloc(n*sizeof(double)));
13         for(i=0;i<n;i++)                         /*读取案例数据*/
14             fscanf(f1,"%lf%lf",x+i,y+i);
15         length=pipe(x,y,n);                      /*计算光束长度*/
16         if(length==DBL_MAX)                      /*穿透*/
17             fprintf(f2,"Through all the pipe.\n");
18         else
19             fprintf(f2,"%.2f\n",length);         /*最长的光束长度*/
20         fscanf(f1,"%d",&n);
21     }
22     free(x);free(y);free(y1);
23     fclose(f1);fclose(f2);
24     return 0;
25 }
```

程序 5-18　解决 PiPe 问题的 C 程序

对程序 5-18 的说明如下。

(1) 第 8～21 行的 while 循环处理输入文件中的每一个案例。其中，第 11 行和第 12 行对 x、y 分配了动态数组空间，第 13 行和第 14 行的 **for** 循环从输入文件中逐一读取各根管子上沿端点坐标，存储于 x、y 中。

(2) 第 15 行调用程序 5-17 定义的函数 pipe 计算本案例管路中最长光线长度 length。

(3) 第 16～18 行的 if-else 语句根据 length 的不同情况向输出文件写入相应的信息。其中，当光线穿透管路时(length 为∞)，第 17 行向输出文件写入信息"Through all the pipe. "，否则第 19 行向输出文件写入 length 的值。

程序 5-15 至程序 5-18 存储在文件夹 chap05/pipe 中的源文件 pipe.c 中，读者可打开文件研读并试运行。

5.5.2　最小边界矩形

Smallest Bounding Rectangle

Given the Cartesian coordinates of $n(>0)$ 2-dimensional points, write a program that computes the area of their smallest bounding rectangle (smallest rectangle containing all the given points).

Input

The input file may contain multiple test cases. Each test case begins with a line containing a positive integer n(<1001) indicating the number of points in this test case. Then follows n lines each containing two real numbers giving espectively the x-and y coordinates of a point. The input terminates with a test case containing a value 0 for n which must not be processed.

Output

For each test case in the input print a line containing the area of the smallest bounding rectangle rounded to the 4th digit after the decimal point.

Sample Input

```
3
3.000 5.000
7.000 9.000
17.000 5.000
4
0.000 10.000
10.000 20.000
20.000 20.000
20.000 10.000
0
```

Sample Output

```
56.0000
200.0000
```

1. 问题描述与分析

本题的题面是很直白的,形式化表述为如下。

输入:平面点集 Q。

输出:平面上涵盖 Q 的最小矩形面积。

为解决此问题,先给出以下结论。

设平面点集 Q 的凸壳为 CH(Q),涵盖 Q 的最小矩形 R 也是涵盖 CH(Q)的最小矩形。并且,至少有 CH(Q)的一条边在 R 的一条边上。

根据这一结论,可以把计算 Q 的最小覆盖矩形转化为计算 CH(Q)的最小覆盖矩形。设 CH(Q)有 h 条边,则计算与 CH(Q)每一条边重合一条边的最小覆盖矩形,求其中的最小者。可以用下列方法计算与 CH(Q)一条边相重的最小覆盖矩形,设 CH(Q)的一条边与 y 轴正向夹角为 θ,则将 CH(Q)的所有点旋转 θ,然后求出旋转得到点集中最小/大横坐标,最小/最大纵坐标,就可得到覆盖 CH(Q)的矩形。

2. 算法描述

SMALLEST-BOUNDING-RECTANGLE(Q)

```
P ←CH(Q)
arer ←∞
for P 的每一条边 s
  do θ ←s 与 y 轴正向夹角
    P′←P 旋转 θ
    x_min, x_max, y_min, y_max ←P′中点的最小/大横/纵坐标
    a ← (x_max − x_min)×(y_max − y_min)
    if a<arer
      then arer ←a
return arer
```

算法 5-9　解决 Smallest Bounding Rectangle 问题的算法过程

3. 程序实现

把矩形表示成如下数据类型。

```
typedef struct{
    double width;              /* 宽度 */
    double height;             /* 高度 */
}Rectangle;
```

1) 平面点的比较

由于在计算过程中需要找到点集中处于最左者、最右者、最高者及最低者，这些点都可以调用程序 5-7 中定义的函数 most 求得。然而，需要为传递给 most 的点的大小比较定义函数。

```
static int pxgreater(Point *a,Point *b){              /* 平面点按横坐标比较大小 */
  if(a->x>b->x)
      return 1;
  if(a->x<b->x)
      return -1;
  return 0;
}
static int pxless(Point *a,Point *b){
  return-pxgreater(a,b);
}
static int pygreater(Point *a,Point *b){              /* 平面点按纵坐标比较大小 */
  if(a->y>b->y)
      return 1;
  if(a->y<b->y)
      return -1;
  return 0;
}
static int pyless(Point *a,Point *b){
  return -pygreater(a,b);
}
```

程序 5-19　平面点按横坐标、纵坐标比较大小的 C 函数

第 1～7 行定义的函数 pxgreater 比较点 a、b 的横坐标，前者大则返回 1；后者大则返回 －1；两者相等则返回 0。第 8～10 行定义的函数 pxless，按与 pxgreater 相反的方向比较点 a、b 的横坐标大小。这只要返回调用 pxgreater 的相反数即可。

第 11～17 行定义的函数 pygreater，以及第 18 行和第 19 行定义的函数 pyless 比较点 a、b 的纵坐标大小，代码与 pxgreater、pxless 的相似，读者可比较研读。

2）平面点集的旋转

```
1  void pointRotate(Point *p,double theta){
2    double x=(p->x)*cos(theta)-(p->y)*sin(theta),
3        y=(p->x)*sin(theta)+(p->y)*cos(theta);
4    p->x=x;
5    p->y=y;
6  }
7  static void chRotate(Point *x,int n,double theta){         /*点集 x 旋转 theta 角*/
8    int i;
9    for(i=0;i<n;i++)
10       pointRotate(x+i,theta);                                /*点 x[i]旋转 theta 角*/
11 }
```

程序 5-20　计算平面点集旋转的 C 函数

对程序 5-20 的说明如下。

平面解析几何告诉我们，若坐标系 xOy 旋转 θ 角形成的新的坐标系记为 $x'Oy'$（见图 5-15）。则 xOy 中一点 (x,y) 在 $x'Oy'$ 中的坐标 (x',y') 满足方程：

$$\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

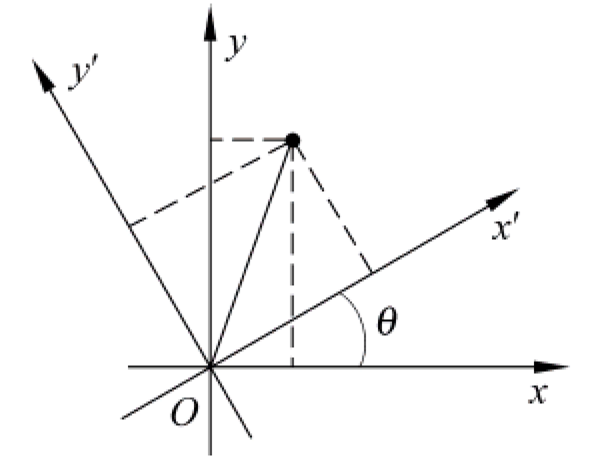


图 5-15　坐标旋转

第 1～6 行定义的函数 pointRotate 就是按照这一方程计算点 p 在旋转了 θ 角后新的坐标系中的坐标。而第 7～11 行定义的函数 chRotate 对存储在数组 x 中的 n 个点，计算旋转 θ 角后新的坐标。其中，第 10 行调用 pointRotate 函数计算第 i 个点的新坐标。

为便于代码重用，将程序 5-20 添加到文件夹 geometry 中的头文件 point.h 和源文件 point.c 中。

3）计算最小覆盖矩形

做好这些准备工作后，我们来实现算法 5-9。

```
Rectangle smallestBoundingRectangle(Point *q,int n){      /*计算点集 q 的最小覆盖矩形*/
  Rectangle result;
  int i,j,m;
  Point *p=grahamScan(q,n,&m),                            /*计算点集 q 的凸包 p*/
      a;
  double minimx,maxmx,minimy,maxmy,area=DBL_MAX,temp;
  for(i=0,j=0;i<m;i++,j++){                    /*对凸包的每一条边计算外接矩形面积*/
      double tg,θ;
```

```
9           a=sub(p+((j+1)%m),p+j);                              /* 一条边 a */
10          tg=a.y/a.x;                                          /* 边 a 的斜率 */
11          theta=PI/2-atan(tg);                                 /* 旋转后边 a 垂直的旋转角 */
12          chRotate(p,m,θ);                                     /* 旋转 */
13          minimx=p[most(p,m,sizeof(Point),pxless)].x;          /* 凸包最左边点横坐标 */
14          maxmx=p[most(p,m,sizeof(Point),pxgreater)].x;        /* 最右边点横坐标 */
15          minimy=p[most(p,m,sizeof(Point),pyless)].y;          /* 最下边点纵坐标 */
16          maxmy=p[most(p,m,sizeof(Point),pygreater)].y;        /* 最上边点纵坐标 */
17          temp=(maxmx-minimx)*(maxmy-minimy);                  /* 覆盖凸包矩形面积 */
18          if(temp<area){                                       /* 跟踪最小者 */
19              area=temp;
20              result.height=(maxmy-minimy);
21              result.width=(maxmx-minimx);
22          }
23      }
24      free(p);
25      return result;
26  }
```

程序 5-21　实现算法 5-9 的 C 函数

对程序 5-21 的说明如下。

(1) 函数 smallestBoundingRectangle 拥有两个参数,数组 q 存储了 n 个 Point 类型的元素。与算法稍有不同,函数返回的是面积最小的覆盖 q 的 Rectangle 数据,而不是矩形的面积值。

(2) 第 7～23 行的 **for** 循环实现算法过程中第 3～9 行的操作,对由第 4 行计算得到的点集 q 的凸包 p 的每一条边计算与此共边的覆盖 q 的矩形,跟踪面积最小者。

(3) 循环体中,第 9 行调用程序 5-1 中定义的函数 sub 计算凸包 p[0..m-1]中两个相邻顶点的差得到线段的向量表示 a。注意由于 p[m-1]与 p[0]相邻,所以两个相邻顶点的地址分别表示为 p+j 和 p+(j+1)%m。第 10 行和第 11 行计算该边需要旋转的角度 θ(其中需要调用库函数 atan 计算反正切),完成算法中第 4 行的操作。第 12 行调用程序 5-20 定义的函数 chRotate,旋转 p 中每一个点,使得线段 a 垂直于新坐标系的横坐标轴,对应算法中第 5 行的操作。第 13～16 行调用程序 5-8 定义的函数 most,分别计算凸包中最左、最右点的横坐标,最高和最低点的纵坐标,完成算法过程中第 6 行的操作。第 8～22 行跟踪面积最小的矩形 result。

为便于代码重用,将程序 5-19 和程序 5-21 存储在文件夹 chap05 中的源文件 overlaprectangle.c 中,将函数 smallestBoundingRectangle 的原型声明存储在头文件 overlaprectangle.h 中。其他函数之所以定义成静态的 **static** 型,是因为它们是仅被 smallestBoundingRectangle 函数所调用的功能函数。

4) **主函数**

利用前面做好的准备工作,编写如下解决 Smallest Bounding Rectangle 问题的程序。

```
1  int main(){
```

```
2    int i,n;
3    FILE *f1=fopen("chap05/SmallestBoundingRectangle/inputdata.txt","r"),
4         *f2=fopen("chap05/SmallestBoundingRectangle/outputdata.txt","w");
5    assert(f1&&f2);
6    fscanf(f1,"%d",&n);
7    while(n){
8        Rectangle r;
9        Point *q;
10       assert(q=(Point*)calloc(n,sizeof(Point)));
11       for(i=0;i<n;i++)
12           fscanf(f1,"%lf %lf",&(q[i].x),&(q[i].y));
13       r=smallestBoundingRectangle(q,n);
14       free(q);
15       fprintf(f2,"%.4f\n",r.width*r.height);
16       fscanf(f1,"%d",&n);
17   }
18   fclose(f1);fclose(f2);
19   return 0;
20 }
```

程序 5-22　解决 Smallest Bounding Rectangle 问题的 C 程序

函数中第 7～17 行的 **while** 循环处理输入文件中的每一个案例。其中,第 10～12 行逐一将本案例中各点坐标读入到数组 q 中,第 13 行调用程序 5-21 定义的函数 smallestBoundingRectangle 计算覆盖点集 q 的面积最小的矩形 r。第 15 行和第 16 行将最小矩形的面积写入输出文件中。

程序 5-22 存储在文件夹 chap05/Smallest Bounding Rectangle 中的源文件 Smallest-BoundingRectangle.c 中,读者可打开文件研读并试运行。

5.5.3　德克萨斯一日游

Texas Trip

Description

After a day trip with his friend Dick, Harry noticed a strange pattern of tiny holes in the door of his SUV. The local American Tire store sells fiberglass patching material only in square sheets. What is the smallest patch that Harry needs to fix his door?

Assume that the holes are points on the integer lattice in the plane. Your job is to find the area of the smallest square that will cover all the holes.

Input

The first line of input contains a single integer T expressed in decimal with no leading zeroes, denoting the number of test cases to follow. The subsequent lines of input describe the test cases.

Each test case begins with a single line, containing a single integer n expressed in decimal with no leading zeroes, the number of points to follow; each of the following n lines contains two integers x and y, both expressed in decimal with no leading zeroes, giving the coordinates of one of your points.

You are guaranteed that $T \leqslant 30$ and that no data set contains more than 30 points. All points in each data set will be no more than 500 units away from (0,0).

Output

Print, on a single line with two decimal places of precision, the area of the smallest square containing all of your points. An answer will be accepted if it lies within 0.01 of the correct answer.

Sample Input

```
2
4
-1-1
1-1
1 1
-1 1
4
10 1
10-1
-10 1
-10-1
```

Sample Output

```
4.00
242.00
```

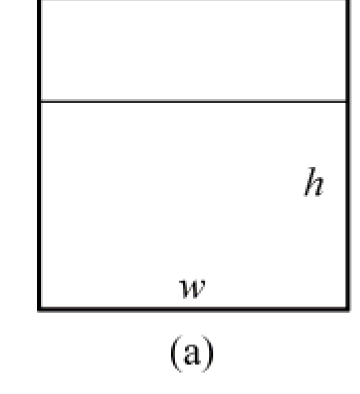
1. 问题描述与分析

Dick 开着 Harry 的 SUV 到德克萨斯州一日游。回来后，Harry 发现车门上有一些由分布细小的点构成的损伤。他准备到汽修店去购买正方形的玻璃胶贴在损伤处，希望找到能覆盖这些小点的最小正方形。此问题的形式化表述如下。

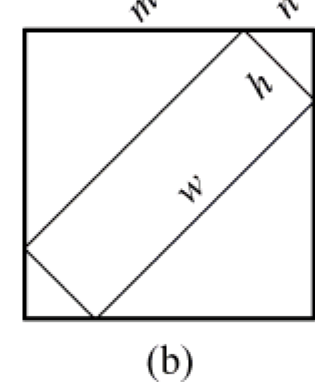
输入：点集 $Q=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$。

输出：覆盖 Q 的最小正方形的面积。

解决这个问题可以利用上一题的结果，先找到覆盖 Q 的最小矩形 R，若 R 已经构成正方形，则返回 R 的面积；若否，则计算覆盖 R 的最小正方形 S 的面积。设 R 的高与宽分别为 h 和 w，不失一般性，假定 $w>h$。根据正方形和矩形的对称性，外接于 R 的最小正方形 S 只能是图 5-16 所示的两种情形之一。



(a)



(b)

图 5-16　外接于 R 的最小正方形

显然，情形 5-16(a)中正方形 S 的面积为 w^2。对

于情形图 5-16(b),R 的角将正方形的每条边分成两段 m 和 n。其中,$2m^2=w^2$,$2n^2=h^2$,即 $m=w/\sqrt{2}$,$n=h/\sqrt{2}$。此时,正方形 S 的面积为 $(m+n)^2=(w+h)^2/2$。

$$(w+h)^2/2 \leqslant w^2 \Leftrightarrow (w+h)^2 \leqslant 2w^2 \Leftrightarrow w+h \leqslant \sqrt{2}w \Leftrightarrow h \leqslant (\sqrt{2}-1)w$$

即当 $h>(\sqrt{2}-1)w$ 时(见图 5-16(a)),覆盖 R 的最小正方形 S 的面积是 w^2;当 $h\leqslant(\sqrt{2}-1)w$ 时(见图 5-16(b)),S 的面积为 $(w+h)^2/2$。

2. 算法描述

在本问题的算法中要调用上一题的 SMALLEST-BOUNDING-RECTANGLE 过程,需要对它做少许修改,返回的不是覆盖点集 Q 的最小矩形 R 的面积,而是该矩形 R 的高 h 与宽 w($h<w$)。

```
TEXAS-TRIP(Q)
R ← SMALLEST-BOUNDING-RECTANGLE(Q)
if R[h] > (√2 − 1) · R[w]
   then return (R[w])²
   else return (R[w] + R[h])²/2
```

算法 5-10 计算覆盖点集最小正方形的过程

3. 程序实现

利用上一个问题中程序 5-21 定义的函数 smallestBoundingRectangle,很容易用 C 语言解决 TexasTrip 问题。

```c
int main(){
    int i,j,n,m;
    FILE *f1=fopen("chap05/TexasTrip/inputdata.txt","r"),
         *f2=fopen("chap05/TexasTrip/outputdata.txt","w");
    assert(f1&&f2);
    fscanf(f1,"%d",&m);
    for(i=0;i<m;i++){
        double w,h,squaroot2=sqrt(2.0);
        Rectangle r;
        Point *q;
        fscanf(f1,"%d",&n);
        assert(q=(Point *)calloc(n,sizeof(Point)));
        for(j=0;j<n;j++)
            fscanf(f1,"%lf %lf",&(q[j].x),&(q[j].y));
        r=smallestBoundingRectangle(q,n);
        free(q);
        w=r.height>r.width? r.height:r.width;
        h=r.height+r.width-w;
        if(h>(squaroot2-1)*w)
            fprintf(f2,"%.4f\n",w*w);
```

```
21          else
22               fprintf(f2,"%.4f\n",(w+h)*(w+h)/2);
23      }
24      fclose(f1);fclose(f2);
25      return 0;
26 }
```

程序 5-23　解决 TexasTrip 问题的 C 程序

程序中第 6 行从输入文件中读取案例数 m。第 7～23 行的 **for** 循环处理输入文件中的每一个案例。其中,第 11 行读取本案例的点数 n,第 12～14 行从输入文件中读取本案例的 n 个点的坐标存储到数组 q 中。第 15 行调用函数 smallestBoundingRectangle,完成算法 5-10 的第 1 行操作,计算覆盖点集 q 的最小矩形 r。第 17 行和第 18 行根据 r 的宽度和高度确定 w 和 h 的值(w≥h)。第 19～22 行的 if-else 语句完成算法 5-10 中第 2～4 行的操作,将计算结果写入输出文件中。

程序 5-23 存储在文件夹 chap05/TexasTrip 中的源文件 TexasTrip.c 中,读者可打开文件研读并试运行。

第6章　数论算法

信息技术广泛深入的应用，对信息的安全要求日益提高。信息安全最基本的技术是密码技术，基于大素数的密码技术将一度被视为一个纯数学课题的数论推到了信息技术应用的前沿。基于大素数的密码方案的可行性依赖于能快速找到一个大素数的能力，而它们的安全性则依赖于对大整数的素因数分解的无奈。本章介绍作为这些应用的基础一些数论理论和相关的算法。

6.1　整数的表示

6.1.1　整数的表示

设 B 为一正整数($B>1$)，对任一正整数 a，在 B 进制下可唯一地表示为

$$a=\sum_{i=0}^{n} a_i B^i,\quad 0\leqslant a_i<B, a_n\neq 0 \tag{6-1}$$

称 a 为 $n+1$ 位的(B 进制)正整数，a_i 称为 a 的第 i 位数字，$i=0, 1, \cdots, n$。例如，当 $B=10$ 时，就是人们最熟悉的十进制整数，而当 $B=2$ 时，就是在计算机中表示的二进制整数。显然 B 越大，相同的位数可表示的数就越大。在计算机中，通常使用的整型数据要受处理器的字长限制。一个字长为 32 位的二进制整型数据的取值范围为 $-2^{31}\sim 2^{31}-1$，即 $-2\,147\,483\,648\sim 2\,147\,483\,647$。若应用中需要更大的取值范围，例如要对 $-2^{1023}\sim 2^{1023}-1$ 范围内的数据进行计算处理，则需要定义更大的整数数据类型。

在定字长计算机中表示长整数的方法往往是将正整数的各位数字顺序表示成一个线性表。我们约定，按从低位到高位顺序存储。一般的整数 a 表示为具有表示其绝对值的线性表 $value[a]=(a_0, a_1, \cdots, a_n)$ 及表示正负符号属性 $sign[a]$ 的对象。整数的算术运算均以绝对值的计算为基础，所以本章主要讨论非负整数的运算。按上述约定，在后面的介绍中一个 B 进制正整数 a 直接表示成 $(a_n a_{n-1}\cdots a_1 a_0)$。

6.1.2　整数的算术运算

1. 加法

正整数 $a=(a_n a_{n-1}\cdots a_1 a_0)$，$b=(b_m a_{m-1}\cdots b_1 b_0)$，它们的和 $a+b$ 记为 $c=(c_t c_{t-1}\cdots c_1 c_0)$。其中，$\max(n, m)\leqslant t\leqslant \max(n, m)+1$，$c_i$ 是 a 与 b 的第 i 位数字 a_i 与 b_i 的和加上低位(第 $i-1$ 位)的数字和对本位的进位除以 B 的余数。写成伪代码过程为

```
ADD(a, b)
```

```
1 carry←0
2 t←min(m, n)
3 for i←0 to t              ▷从低位到高位逐位计算
4   do c_i←a_i+b_i+carry
5      carry← c_i/B
6      c_i← c_i mod B
7 for i←t+1 to n            ▷对 n>m 的情形
8   do c_i← a_i+carry
9      carry← c_i/B
10     c_i← c_i mod B
11 for i←t+1 to m           ▷对 m>n 的情形
12  do c_i← b_i+carry
13     carry← c_i/B
14     c_i← c_i mod B
15 if carry≠0
16   then c_i←carry
17 return c
```

算法 6-1　计算 B 进制正整数 a 与 b 的和的过程

过程 ADD 运行如下。第 3～6 行的 **for** 循环对 a 与 b 从低位到高位逐位进行加法计算，其中第 4 行计算本位数 a_i 与 b_i 的和并加入上一位数字和对本位的进位 *carry*。*carry* 在第 1 行初始化为 0，这是因为刚开始对最低位计算和时，没有前一位的进位。第 5 行计算本位对下一位的进位，记录在 *carry* 中，第 6 行完成和的本位值 c_i 的计算。对 a、b 的位数不同的情形，需要对较长的部分做处理：每一位要与前一位的进位相加。这由第 7～10 行及第 11～14 行的 **for** 循环完成。第 15～16行根据最高位是否有进位决定数位是否有增加。

若将 a、b 的较大位数记为 n，不难看出 ADD 过程的运行时间是 $\Theta(n)$。

2. 减法

设 B 进制正整数 $a=(a_na_{n-1}\cdots a_1a_0)$，$b=(b_ma_{m-1}\cdots b_1b_0)$，$a\geqslant b$。$a$ 与 b 的差 $a-b$ 记为 $c=(c_tc_{t-1}\cdots c_1c_0)$。其中，$0\leqslant t\leqslant n$，当 $a_i\geqslant b_i$ 时，$c_i=a_i-b_i$；而当 $a_i<b_i$ 时，$c_i=a_i-b_i+B$，此时 a_{i+1} 减小 1。写成伪代码过程如下。

```
SUB(a, b)                 ▷计算 a−b，假定 a≥b
for i←0 to m
  do c_i←a_i−b_i
    if c_i<0
     then c_i←c_i+B
         a_(i+1)← a_(i+1)−1
for i←m+1 to n
  do c_i←a_i
    if c_i<0
      then c_i←c_i+B
           a_(i+1)← a_(i+1)−1
消去 c 的高位 0
```

```
12 return c
```

算法 6-2　计算 B 进制正整数 a、b 的差过程

很明显,算法中第1~5行的 **for** 循环重复 m 次,而第6~10行的 **for** 循环重复 $n-m$ 次,故算法的运行时间为 $\Theta(n)$。

3. 乘法

为说明长整数的乘法,先来看下面的两个十进制整数相乘的"竖式"。

```
          3  1  2  4
       ×     2  1  3
       --------------
          9  3  7  2
       3  1  2  4
 --------------------
       4  0  6  1
 +  6  2  4  8
 --------------------
    6  6  5  4  1  2
```

其中 $a=(a_3a_2a_1a_0)=(3124)$,$b=(b_2b_1b_0)=(213)$,将 $c=(c_6c_5c_4c_3c_2c_1c_0)$ 初始化为0。考察第1根横线下的第1行数据(9 3 7 2),分别是 a_3、a_2、a_1、a_0 与 $b_0=3$ 相乘对应积与 c_0、c_1、c_2、c_3(均为0)的和。第2根横线下第1行数据4 0 6 1,是 a_3、a_2、a_1、a_0 与 $b_1=1$ 相乘对应积(3 1 2 4)分别与 c_4、c_3、c_2、c_1(0 9 3 7)的和。而第3根横线下的一行数据(6 6 5 4 1 2)就是积($c_5c_4c_3c_2c_1c_0$)$=c=ab$。其中,(6 6 5 4)是 a_3、a_2、a_1、a_0 与 $b_2=2$ 相乘对应的积(6 2 4 8)与 c_5、c_4、c_3、c_2(0 4 0 6)的和。将此过程写成伪代码如下。

```
PRODUCT(a, b)
1 for i←0 to m
2   do carry←0
3      for j←0 to n
4        do c[i+j]←b[i]×a[j]+ c[i+j]+carry
5           carry←c[i+j]/B
6           c[i+j]←c[i+j] mod B
7           if carry≠0
8              then c[i+j]←carry
9 return c
```

算法 6-3　计算两个正整数 a、b 的积的过程

PRODUCT 过程的第1~8行的 **for** 循环从高位到低位逐位按上述讨论的方法计算积。若 a、b 都是 $n+1$ 位 B 进制数,则过程的运行时间为 $O(n^2)$。

4. 带余除法

首先,不证明地罗列自然数的一个基本性质。

最小数原理　若 M 是自然数集 $\mathbf{N}$ 的任一非空子集(有限或无限均可),则 M 中必有最小的数。

我们将会看到,这条关于自然数的基本原理,是本章所讨论的全部理论的出发点。

定理 6-1(除法定理)　对任意整数 a 和正整数 n,存在唯一的整数 q 和 r 使得

$$a = qn + r, 0 \leqslant r < n \tag{6-2}$$

证明　仅对 a 是自然数的情形给出证明,对于 a 为负整数的情况很容易由自然数的结论加以推导。构造集合 $M=\{q' \in \mathbf{N}: q'n \geqslant a\} \subseteq \mathbf{N}$,根据自然数的最小数原理知,$M$ 中有一个最小数 $q'_{\min}$。

(1) 若 $q'_{\min}n=a$,则取 $q=q'_{\min}$,$r=0$ 即为唯一满足本定理中条件的 q 和 r。

(2) 若 $q'_{\min}n>a$,取 $q=q'_{\min}-1$,$r=a-qn$。则有 $qn<a$,且 $0<r=a-(q'_{\min}-1)n=a-q'_{\min}n+n=n-(q'_{\min}n-a)<n$。而 $qn+r=qn+(a-qn)=a$。由自然数的算术运算的唯一性知,此 q 和 r 是唯一存在的。

合并(1)和(2)就得到了本定理的证明。

对给定的整数 a 和正整数 n,计算定理 6-1 中值 q 和 r 的运算就是人们所熟悉的整数的除法 a/n。把值 q 称为 a 除以 n 的**商**,记为$\lfloor a/n \rfloor$。值 r 称为 a 除以 n 的余数或 a 关于模 n 的**余数**,记为 $a \bmod n$。当 a 除以 n 的余数 $a \bmod n=0$ 时我们称 n **整除** a。

以十进制数为例,考察整数的除法算法。先考虑除数为 1 位数的正整数除法,例如,$a=(a_2a_1a_0)=327$,$b=(b_0)=2$,计算 a/b 的竖式如下。

```
     1  6  3
  ──────────
2)  3  2  7
    2
  ──────────
    1  2
    1  2
  ──────────
          7
          6
  ──────────
          1
```

式中商 $q=163$。若将余数 r 初始化为 0,$q_2=1=3/2=(rB+a_2)/b_0$,此时 $r=(rB+a_2) \bmod b_0=3 \bmod 2=1$,$rB+a_1=1\times10+2=12$ 恰为第 1 根横线下的数。$q_1=6=12/2=(rB+a_1)/b_0$,此时 $r=(rB+a_1) \bmod b_0=12 \bmod 2=0$,$rB+a_0=0\times10+7=7$ 恰为第 2 根横线下的数。$q_0=3=7/2=(rB+a_0)/b_0$,此时 $r=(rB+a_0) \bmod b_0=7 \bmod 2=1$。如此,$q=(q_2q_1q_0)$。将此过程写成伪代码如下。

```
DIVIDED-BY-DIGIT(a, b)   ▷除数 b 是 1 位正整数
r←0
for j←n downto 0
    do qj←⌊(rB+aj)/b⌋
       r←(rB+aj) mod b
return q and r
```

算法 6-4　计算 B 进制正整数 a 除以 1 位 B 进制正整数 b 的过程

显然,过程的运行时间是 $\Theta(n)$。其中,n 是 a 的位数。

下面考虑除数是多位数的情况。设 $a=327$,$b=23$,则 a、b 相除的竖式如下。

```
          1  4
      ----------
2  3) 3   2  7
      2   3
      ----------
          9  7
          9  2
          ------
             5
```

在上式中,$a=(a_2a_1a_0)=(327)$,$b=(b_1b_0)=(23)$。商 $q=a/b$ 的位数至多为 $n-m+1=2-1+1=2$。而余数$r=a \bmod b$ 的位数至多为 $m=2$。与除数 b 为 1 位数的情形相仿,将中间余数 r 初始化为 $3=a_2$,令 $q_1=1=32/23=(r*10+a_1)/b$。此时,$(r*10+a_1) \bmod b=32 \bmod 23=9$,令其为新的 r。接下来令 $q_0=4=97/23=(r*10+a_0)/b$。此时,$(r*10+a_0) \bmod b=97 \bmod 23=5$,令其为新的 r。则 $q=(q_1q_0)=14$,即为商 a/b,而 $r=5$ 即为 a 除以 b 的余数。一般地,对于 $n+1$ 位正整数 $a=(a_na_{n-1}\cdots a_1a_0)$,$m+1$ 位正整数 $b=(b_mb_{m-1}\cdots b_1b_0)$,为计算式(6-2)中的 q 和 r,将 r 初始化为$(a_na_{n-1}\cdots a_{n-m+1})$。$j$ 从 $n-m$ 起到 0 为止,依次计算 $q_j=(r\cdot B+a_j)/b$,重置 $r=(r\cdot B+a_j) \bmod b$(即置 $r=(r\cdot B+a_j)-q_j\cdot b$)。则 $q=(q_{n-m}q_{n-m-1}\cdots q_1q_0)$和 r 即为所求。需要注意的是此处 b 不是 1 位数,而是 $m+1$ 位数。计算 $q_j=(r\cdot B+a_j)/b$,$j=n-m,\ n-m-1,\ \cdots,\ 0$ 是除法中的“智慧”所在。最直接的方法是从 0 到 $B-1$ 逐一检测,直至 $0\leqslant(r\cdot B+a_j)-q_j\cdot b\leqslant b$。当 B 较大时,这是比较费时的。下列引理使得我们能尽快地找到合适的 q_j。注意$(r\cdot B+a_j)$最多比 b 多 1 位。

引理 6-1 B 进制正整数 a、b 都是 n 位数且 $a\geqslant b$。若 $b_{n-1}\geqslant\lfloor B/2\rfloor$,则 $q=a_{n-1}/b_{n-1}$。

引理 6-2 设 B 进制正整数 $a=(a_n\cdots a_1a_0)$,$b=(b_{n-1}\cdots b_1b_0)$,$a>b$ 且$(a_{n-1}\cdots a_1a_0)<b$。记$\hat{q}=\min\left(\left\lfloor\frac{a_nB+a_{n-1}}{b_{n-1}}\right\rfloor,B-1\right)$,若 $b_{n-1}\geqslant\lfloor B/2\rfloor$,则商 $a/b=q$ 必有$\hat{q}-2\leqslant q\leqslant\hat{q}$。

引理 6-1 和引理 6-2 告诉我们,在 a 至多比 b 多 1 位及 $b_{n-1}\geqslant\lfloor B/2\rfloor$的前提条件下,即至多做 3 次试商即可得到商 $q=\lfloor a/b\rfloor$。而条件 $b_{n-1}\geqslant\lfloor B/2\rfloor$可以通过除法前 a 和 b 同时乘以 $d=\lfloor B/(b_{n-1}+1)\rfloor$而得以保证(这一操作称为“规格化”),除法后,将 r 除以 d(去规格化)而得。

```
DIVISION(a, b)              ▷对 n+1 位及 m+1 位正整数 a,b 计算式(6-2)中的 q 和 r
if a<b
  then q←0, r←a
       return q and r
if a=b
  then q←1
       r←0
       return q and r
if b_m≤B/2                  ▷计算规格化因子 d
  then d←⌊B/(b_m+1)⌋
  else d←1
u←da, v←db, n← u 的位数−1, m←v 的位数−1    ▷规格化
r←(u_n u_{n−1}…u_{n−m+1})
for j←n−m downto 0
  do if r<v
```

15　　**then** $q_j \leftarrow 0$
16　　**else if** r 的位数 $=v$ 的位数
17　　　　**then** $q_j \leftarrow r_m / v_m$
18　　　　　**else** $q_j \leftarrow \min(\lfloor (u_j B + u_{j-1})/v_m \rfloor, B-1)-2$
19　　$r \leftarrow r - q_j v$
20　　**while** $r \geqslant v$
21　　　**do** $r \leftarrow r - v$
22　　　　$q_j \leftarrow q_j + 1$
23　　$r \leftarrow rB + a_j$
24 $r \leftarrow r/d$　　　　　　▷去规格化
25 去除 q 的高位 0
26 **return** q and r

算法 6-5　计算 B 进制正整数 a 除以 b 的商与余数的过程

算法 DIVISION 的第 1～3 行处理 $a<b$ 的情形，此时 $q=0$，$r=a$。第 4～7 行处理 $a=b$ 的情形，此时 $q=1$，$r=0$。第 8～10 行计算规格化因子。第 11 行对 a，b 作规格化操作。第 12 行初始化中间余数 r。第 13～23 行的 **for** 循环从高位到低位逐位计算商。其中，第 14～18 行针对 3 种不同的情形——$r<v$、$r \geqslant v$ 且 r、v 的位数相等及 r 比 v 多 1 位——初始化商的第 j 位 q_j。第 19 行计算中间余数 r 与 $q_j v$ 的差，第 20～22 行的 **while** 循环确定 q_j。根据引理 6-1 和引理 6-2 知，该循环至多重复 3 次。第 23 行计算新的中间余数 r。完成上面的计算后，第 24 行执行去规格化操作。第 25 行去除 q 中可能的高位 0。

当 $n=2m$ 时，第 13～23 行的 **for** 循环重复 m 次，循环体中第 19 行的运算耗时 $\Theta(m)$，故 DIVISION 的运行时间为 $\Theta(m^2)$。

6.1.3　程序实现

1. 大整数的数据类型定义及常规维护

本章开发一个基数为 10^9 的大整数类型。由于在整数的运算过程中所得到的结果位数是不能事先预见的，所以用一个链表来存储长整数的各位数字是合适的。还要用一个整型数据表示长整数的符号，约定：0 表示正数，1 表示负数。

```
#define Base 1000000000              /* 大整数的基数 */
typedef struct{
    LinkedList * value;              /* 从低位到高位存储大整数的绝对值 */
    int sign;                        /* 大整数符号：0 为正，1 为负 */
}BigInt;
BigInt newIntBystring(char * s){   /* 用表示大整数的串创建大整数 */
    int n,i;
    long x;
    div_t qr;                        /* C 语言整除类型数据 quot 属性表示商，rem 属性表示余数 */
    BigInt a;
    char digit[10];
```

```
12   a.value=createList(sizeof(long),NULL);
13   if(s[0]=='-'){                          /*创建一个负的大整数*/
14        a.sign=1;
15        s++;                               /*掠过负号*/
16   }else                                   /*正整数*/
17      a.sign=0;
18   n=strlen(s);                            /*十进制位数*/
19   qr=div(n,9);                            /*每 9 位十进制数字构成一位 Base 进制数字*/
20   if(qr.rem){                             /*Base 进制最高位含十进制数字位数 rem*/
21       strncpy(digit,s,qr.rem);            /*读取最高 rem 位*/
22       x=strtoul(digit,NULL,10);           /*转换成十进制整数*/
23       listPushFront(a.value,&x);          /*置为 Base 进制最高位*/
24   }
25   for(i=0;i<qr.quot;i++){                 /*对 quot 位 Base 进制数字逐位处理*/
26       strncpy(digit,s+qr.rem+i*9,9);      /*读取 9 位十进制数字*/
27       x=strtoul(digit,NULL,10);           /*转换成整数*/
28       listPushFront(a.value,&x);          /*置为 Base 进制当前最低位*/
29   }
30   return a;
31 }
32 void printInt(BigInt *a){                 /*屏幕输出大整数(Base 进制)*/
33   ListNode *p;
34   if(listEmpty(a->value))
35       return;
36   if(a->sign)                             /*负数*/
37       printf("-");
38   p=a->value->nil->prev;                  /*Base 进制最高位*/
39   printf("%lu",*((long*)(p->key)));
40   p=p->prev;
41   while(p!=a->value->nil){                /*自高向低逐位输出*/
42       printf("%09lu",*((long*)(p->key)));
43       p=p->prev;
44   }
45 }
```

程序 6-1　大整数类型定义及常规维护函数

对程序 6-1 的说明如下。

(1) 第 1 行定义了一个值为 10^9 的宏 Base,表示长整数的基数。第 2～5 行将长整数类型定义成结构体 BigInt,它具有两个属性:表示绝对值的链表(程序 2-1 中定义的 LinkedList 类型)value 和表示符号的整数 sign。

(2) 第 6～31 行定义的函数 newIntBystring 用传递给它的表示十进制长整数的字符串 s 创建并返回一个 BigInt 对象。s 里面存储的是诸如"12345"或"-12345"这样的表示整数的字符串。函数的设计思想并不困难:以下例说明。对表示十进制长整数的串

"31284763 285792454 284512541 876491874 291631524"

创建一个基数 Base 为 10^9 的长整数，首先调用库函数 div 将串的长度 n=44 除以 9，商 4 余 8。这意味着将其转换成基数为 10^9 的长整数有 5 位(用空格隔开)，除最高位为十进制 8 位数外，其余 4 位为 9 位十进制数。

接下来对 s 从头开始调用库函数 strncpy 截取长度为 8 的子串，调用库函数 strtoul 将其转换成整数，作为长整数的最高位数字。然后循环 4 次每次从串的当前位置开始截取长度为 9 的子串，转换成整数，调用程序 2-4 中定义的函数 listPushFront 将此整数插入到链表 value 中，自高向低地置为长整数的各位数字。

(3) 第 32～45 行定义的函数 printInt 将在屏幕当前位置输出传递给它的参数 a 所指引的长整数。由于长整数的各位数字是按自低向高的顺序存储在链表 value 中。所以，程序是自链表的尾结点开始依次输出 a 的绝对值的各位数字。打印每一位 10^9 进制的数字，需要输出 9 位十进制数，为了对不足 9 位的数也能正确输出，要用格式符“%09d”来对高位空白填 0。

此外，第 19 行调用的库函数 div 的原型声明为

```
div_t div( int numerator, int denominator );
```

该函数计算参数 numerator 除以参数 denominator 的商 quo 和余数 rem。quo 和 rem 整合为系统定义的结构提类型 div_t 的两个成员属性。

第 21 行和第 26 行调用的库函数 strncpy 的原型声明为

```
char * strncpy( char * to, const char * from, size_t count );
```

该函数将参数 from 指引串中由参数 count 确定的长度的子串复制到由参数 to 指引的串中。

第 22 行和 27 行调用的函数 strtoul 的原型声明为

```
unsigned long strtoul( const char * start, char * * end, int base );
```

该函数将由参数 start 指引的串中按参数 base 表示的进位制整数的串转换成长整型数据返回。其中参数 end 将指出 start 中表示整数数据的子串的末尾。若应用中无须此信息，可传递 NULL，如在此处的用法。

函数 div 及 strtoul 的原型声明于头文件 stdlib.h 中，strncpy 的原型声明于头文件 string.h 中。

2. 大整数的算术运算

长整数的算术运算的基础操作是针对绝对值的，即针对正整数的。算法 6-1～算法 6-5 都是针对正整数设计的。对一般的大整数的算术运算，根据整数运算的代数法则(符号法则)，调用该算法即可求得。限于篇幅，仅列出大整数加法的 C 语言实现代码并加以解析。

1) **正整数的加法**

首先实现计算正整数加法的算法 ADD。

```
LinkedList * valueAdd(LinkedList * a, LinkedList * b){
    LinkedList * x, * y, * z=createList(sizeof(long), NULL);
```

```
3    ListNode *p, *q;
4    long r,carry=0;
5    if(a->n>=b->n){                  /* x 为 a、b 中较高位者 */
6        x=a;y=b;
7    }else{
8        x=b;y=a;
9    }
10    p=x->nil->next;q=y->nil->next;  /* p、q 分别指向 x、y 的最低位 */
11    while(q!=y->nil){
12        r= *((long *)(p->key))+ *((long *)(q->key))+carry;
13        carry=r/Base;               /* 向高位的进位 */
14        r=r%Base;                   /* 本位和 */
15        listPushBack(z,&r);         /* 插入和的当前位 */
16        p=p->next;q=q->next;
17    }
18    while(p!=x->nil){               /* x 还有高位未完成运算 */
19        r= *((long *)(p->key))+carry;
20        carry=r/Base;
21        r=r%Base;
22        listPushBack(z,&r);
23        p=p->next;
24    }
25    if(carry)                       /* 向最高位有进位 */
26        listPushBack(z,&carry);
27    return z;
28 }
```

程序 6-2 实现算法 6-1 ADD 过程计算两个正整数加法的 C 函数

对程序 6-2 的说明如下。

(1) 函数 valueAdd 实现算法 6-1 的 ADD 过程,对由指针参数 a、b 指引的表示成链表的大整数绝对值进行加法运算,返回表示成链表的和。

(2) 函数在第 2 行设置 3 个链表指针 x、y、z。x、y 分别指向 a、b 中较长者和较短者。z 指向 a、b 的和。第 3 行设置了 2 个链表结点指针 p、q,分别指向 x、y 的当前结点,即两个正整数的当前位。第 4 行设置的 **long** 型变量 r 和 carry 分别用来表示每一位的和及向高位的进位值。carry 初始化为 0。

(3) 第 5～9 行的 **if-else** 结构完成 x、y 的正确指向(x 指向 a、b 中较长者,y 指向较短者)。第 10 行将 p、q 分别初始化为 x、y 的最低位指针。第 11～17 的 **while** 循环实现算法 6-1 中第 3～6 行的 **for** 循环,从个位开始计算较低位的和。第 18～24 行的 **while** 循环实现算法中第 7～10 行和第 11～14 行的 **for** 循环,完成对较高位的和的计算。第 25 行和第 26 行的 **if** 语句,实现算法中的第 15 行和第 16 行的 **if-then** 结构。对和的最高位进行处理。

注意,在处理过程中,向表示和的 z 插入每一位的操作,都是调用程序 2-4 中定义的函数 listPushBack 完成的。这是因为,曾经约定长整数的绝对值是按从低位到高位的顺序存储的。

2）**正整数的减法**

下面来实现计算正整数减法的算法 SUB。

```
1 LinkedList* valueSub(LinkedList *a, LinkedList *b){
2   LinkedList *x=listClone(a), *y=b, *z=createList(sizeof(long), NULL);
3   ListNode *p=x->nil->next, *q=y->nil->next;
4   long r;
5   while(q!=b->nil){
6       r=*((long*)(p->key))-*((long*)(q->key));
7       p=p->next;q=q->next;
8       if(r<0){
9           r+=Base;
10          (*((long*)(p->key)))--;
11      }
12      listPushBack(z,&r);
13  }
14  while(p!=x->nil){
15      r=*((long*)(p->key));
16      p=p->next;
17      if(r<0){
18          r+=Base;
19          *((long*)(p->key))--;
20      }
21      listPushBack(z,&r);
22  }
23  p=z->nil->prev;
24  while(p!=z->nil->next&&*((long*)(p->key))==0){
25      listDelete(z,p);
26      p=z->nil->prev;
27  }
28  clrList(x,NULL);
29  free(x);
30  return z;
31 }
```

程序 6-3　实现算法 6-2 SUB 过程计算两个正整数减法的 C 函数

对程序 6-3 的说明如下。

（1）函数 valueSub 实现算法 6-2 的 SUB 过程，对由指针参数 a、b 指引的表示成链表的大整数绝对值进行减法运算，返回表示成链表的差。此处，假定传递进来的两个大整数a≥b。

（2）函数在第 2 行设置 3 个链表指针 x、y、z。由于在数的减法过程中，被减数的某一位可能要借位给低位，而改变被减数原来的值，所以需要将被减数 a 复制到 x 中。y 直接指向 b。z 指向 a、b 的差。第 3 行设置了 2 个链表结点指针 p、q，分别指向 x、y 的当前结点，即两个正整数的当前位，初始化为 x、y 的最低位指针。第 4 行设置的 **long** 型变量 r 用来表示每

一位的差。

(3) 第 5～13 行的 **while** 循环实现算法 6-2 中第 1～5 行的 **for** 循环,从个位开始计算较低位的差。第 14～22 行的 **while** 循环实现算法中第 6～10 行的 **for** 循环,完成对较高位的差的计算。由于两个整数相减可能造成高位置 0,第 23～27 行实现算法中的第 11 行操作,消除差的高位 0。

3) **正整数的比较**

```
1 int valueCompare(LinkedList *a, LinkedList *b){
2   ListNode *p=a->nil->prev, *q=b->nil->prev;
3   if(a->n>b->n)
4       return 1;
5   if(a->n<b->n)
6       return -1;
7   while(p!=a->nil){
8       if(*((long *)(p->key))>*((long *)(q->key)))
9           return 1;
10      if(*((long *)(p->key))<*((long *)(q->key)))
11          return -1;
12      p=p->prev;q=q->prev;
13  }
14  return 0;
15 }
```

程序 6-4 大正整数大小比较函数

函数 valueCompare 对由参数 a、b 指引的两个大正整数比较大小。若前者大,返回 1;若两者相等,则返回 0;若前者小,则返回−1。函数体中的第 3 行和第 4 行对应 a 的位数大于 b 的位数的情况,第 5 行和第 6 行处理相反情况。第 7～13 行处理 a、b 位数相等的情况,从高位到低位逐位检测,遇到不相等的情形,随即决定大小。若所有位都相等,第 14 行返回 0。

4) **大整数的和**

利用程序 6-2～程序 6-4,下列函数计算两个大整数的和。

```
BigInt sum(BigInt a, BigInt b){
  BigInt c;                                   /* a、b 的和 */
  int comp=valueCompare(a.value,b.value);     /* a、b 的绝对值比较 */
  if(a.sign==b.sign){                         /* a、b 符号相同 */
      c.value=valueAdd(a.value,b.value);
      c.sign=a.sign;
      return c;
  }
  if(comp==0){                                /* 符号相反,绝对值相同 */
      long x=0;
      c.value=createList(sizeof(long),NULL);
      listPushBack(c.value,&x);
      c.sign=0;
```

```
14        return c;
15    }
16    if(comp>0){                                  /*符号相反,a的绝对值大于b的绝对值*/
17        c.value=valueSub(a.value,b.value);
18        c.sign=a.sign;
19    }else{                                       /*符号相反,b的绝对值大于a的绝对值*/
20        c.value=valueSub(b.value,a.value);
21        c.sign=b.sign;
22    }
23    return c;
24 }
```

程序 6-5　计算两个大整数和的 C 函数

对程序 6-5 的说明如下。

(1) 函数 sum 计算由参数 a、b 表示的两个大整数(可正可负)的和并返回。

(2) 函数中设置两个变量,一个是表示和的 BigInt 型数据 c,另一个是表示 a、b 的绝对值大小比较结果的整型数据 compare。第 3 行 compare 初始化为调用程序 6-4 定义的函数 valueCompare,比较 a、b 的绝对值的返回值。

(3) 第 4～8 行的 **if** 语句检测 a、b 符号相同的情形。第 9～15 行的 **if** 语句检测 a、b 反号绝对值相等的情形。第 16～22 行的 **if-else** 语句检测 a、b 反号绝对值不等的情形。处理的原则是同号绝对值相加符号不变,反号绝对值相减符号按绝对值大的设置。

为便于代码管理,将程序 6-2 和程序 6-3 的函数原型声明于文件夹 numbertheory 中的头文件 valuecalc.h 中,函数定义于同一文件夹的源文件 valuecalc.c 中。为便于代码重用,程序 6-4 的函数原型声明于文件夹 numbertheory 中的头文件 bigint.h 中,而函数定义于同一文件中的源文件 bigint.c 中。

6.1.4　应用

The X Problem

Describtion

We are not alone.

In the year 3000, scientists have eventually found another planet which has intelligent being living on. Let's say, Planet X. And we call the intelligent being there "Xmen". To learn advanced technology from Planet X, we want to exchange information with XMen. Unfortunately, XMen use a very strange data format when sending message, which is called m-encoding (m is for multiplication). Scientists on earth have successfully found out the algorithm of m-encoding, defining as following:

For each data package from XMen, there are two non-negative integers, A and B. And the actual data XMen want to send is the product of A and B ($A*B$).

So, in this problem, you are to implement a decoder for data packages from XMen.

Input

There are multiple tests. Input data starts with an integer N, indicating the number of test cases.

Each test case occupies just one line, and contains two non-negative integers A and B ($0 \leqslant A, B \leqslant 10^{1000}$), separated by just one space.

Output

For each test case, output the actual data XMen want to send, one in a line.

Sample Input

```
3
1 1
100 123
12345789012345678900 54321
```

Sample Output

```
1
12300
670635604939629623526900
```

人类试图与 X 星球上的 X 人通信。X 人发出来的信息是经过编码的:表示成两个不超过 10^{1000} 的非负整数 A 和 B,解码需要将 A、B 相乘得到积 $A \times B$。由于超过了计算机系统的定长整型数据的取值范围,所以需要用到 BigInt 类型。程序代码如下。

```
int main(){
 int n,i;
 FILE *f1=fopen("chap06/TheXProblem/inputdata.txt","r"),   /*输入文件*/
      *f2=fopen("chap06/TheXProblem/outputdata.txt","w");/*输出文件*/
 assert(f1&&f2);
 fscanf(f1, "%d", &n);                                      /*读取案例数*/
 for(i=0;i<n;i++){                                          /*处理每个案例*/
     char a[1000], b[1000], *s;
     BigInt A, B, C;
     fscanf(f1, "%s%s", a, b);                              /*读取 a、b*/
     A=newIntBystring(a); B=newIntBystring(b);              /*生成大整数 A、B*/
     C=product(A, B);                                       /*C←A*B*/
     s=toString(&C);                                        /*转换为串*/
     fprintf(f2,"%s\n",s);                                  /*写入输出文件*/
     free(s);
     clrBigInt(&A);clrBigInt(&B);clrBigInt(&C);
  }
  fclose(f1);fclose(f2);
  return 0;
}
```

程序 6-6 解决 The X Problem 问题的 C 程序

程序 6-6 的说明如下。

(1) 第 3～5 行分别打开输入、输出文件 f1 和 f2。第 6 行读取输入文件中的测试案例数 n。

(2) 第 7～17 行的 **for** 循环处理每一个测试案例。对每一个案例,第 10 行从 f1 中读取表示整数 A、B 的值的 a、b。由于 A、B 是大整数,所以不能直接表示成整型数据,而要通过串 a、b 创建 BigInt 对象 A 和 B(第 11 行)。第 12 行调用计算大整数积的函数 product 计算 A * B,将函数值(积)赋予 C。第 13 行调用函数 toString 将 C 的值转换为串,并赋予 s。第 14 行将 s 作为一行写入 f2。第 16 行调用函数 clrBigInt 清理 A、B、C 的空间,以防内存泄漏。

(3) 第 12 行调用的函数 product 实现计算整数乘法的算法 6-3。第 13 行调用的函数 toString 的定义类似于程序 6-1 中定义的 printInt,不过是把对库函数 printf 的调用改变成对库函数 sprintf 的调用。第 16 行调用的函数 clrBigInt 负责清理由参数指引的大整数的存储空间。这些函数都定义于源文件 bigint. c 中,读者可打开文件夹 numbertheory 中的该文件研读。

6.2　初等数论的概念

本节介绍关于整数集 **Z**={…, −2, −1, 0, 1, 2,…}和自然数集 **N**={0, 1, 2, …}上的初等数论的一些基本概念。

1. 整除性和约数

一个整数能被另一个整数整除的概念是数论的核心概念之一。符号 $d \mid a$(读作“d 整除 a”)意为有某个整数 k,使得 $a=kd$。每个整数整除 0。若 $a>0$ 且 $d \mid a$,则 $|d| \leqslant |a|$。若 $d \mid a$,则说 a 是 d 的一个**倍数**。若 d 不能整除 a,记为 d　a。注意 $d \mid a$ 当且仅当 $-d \mid a$,由于 a 的一个约数的相反数也是 a 的约数,所以,不失一般性,定义约数是非负的,即若 $d \mid a$ 且 $d \geqslant 0$,就说 d 是 a 的一个**约数**。一个整数 a 的约数至少为 1 但不大于 $|a|$。例如,24 的约数有 1、2、3、4、6、8、12 和 24。

每个整数 a 都能被**平凡约数** 1 和 a 整除。a 的非平凡约数也称为 a 的**因数**。例如,20 的因数是 2、4、5 和 10。

2. 余数和模等价类

给定整数 n,全体整数可按被 n 除的不同余数加以划分。定理 6-1 是这一分类的基础。

由于对给定的正整数 n,对于任意整数 a,根据定理 6-1,都有唯一的 a 除以 n 的余数 r,且 $0 \leqslant r < n$,即 r 只可能是集合{0, 1, 2, …, $n-1$}中的一个元素。如果两个整数 a 和 b 除以 n 的余数相同,称 a 和 b 关于模 n **同余**,记为 $a \equiv b \pmod n$。显然,$a \equiv b \pmod n$ 当且仅当存在整数 k,使得 $a-b=kn$。

整数间关于模 n 同余的关系是一个等价关系,即关系≡满足传递性、对称性和自反性。所以,可以利用同余关系将整数集合 **Z** 进行划分。把所有关于模 n 的余数为 $r(0 \leqslant r < n)$的

整数构成的集合称为模 n 的一个**同余类**。所以,整数集 $\mathbf{Z}$ 共有 n 个关于模 n 的同余类。含有整数 a 的模 n 的同余类记为

$[a]_n=\{a+kn : k\in\mathbf{Z}\}$。

例如,$[3]_7=\{\cdots, -11, -4, 3, 10, 17, \cdots\}$;此集合也可表示为$[-4]_7$,$[10]_7$,…。注意,写法 $a\in[b]_n$ 和 $a\equiv b \pmod n$是一样的。所有这样的同余类构成的集合为

$$\mathbf{Z}_n = \{[a]_n : 0\leqslant a\leqslant n-1\} \tag{6-3}$$

为表达方便,常用记号

$$\mathbf{Z}_n = \{0, 1, \cdots, n-1\} \tag{6-4}$$

来表示式(6-3),将其中的 0 理解为$[0]_n$,1 理解为$[1]_n$ 等,将集合 $\mathbf{Z}_n$ 中每一个类表示成其中最小的非负元素。要记住同余类的本质。例如,对 $\mathbf{Z}_n$ 中-1 的引用就是对$[n-1]_n$ 的引用,因为$-1\equiv n-1 \pmod n$。

关于整数的同余关系有如下的常用性质。

若 $a\equiv b \pmod n$,则有如下关系。

(1) 对任意的整数 c,有

$$a+c\equiv b+c \pmod n。\tag{6-5}$$

(2) 对任意的整数 c,有

$$ac\equiv bc \pmod n。\tag{6-6}$$

(3) 对任意的正整数 c,有

$$ac\equiv bc \pmod{nc}。\tag{6-7}$$

(4) 若还有

$$x\equiv y \pmod n,则\ ax\equiv by \pmod n。\tag{6-8}$$

为说明本章以后若干重要结论,在此证明一个关于整数同余关系的重要性质。

定理 6-2 设 a 和 b 是满足 $a\mid b$ 和 $b>0$ 的整数。对任意的整数 x 和 y,$x\equiv y \pmod b$ 蕴涵着 $x\equiv y \pmod a$。

证明 由 $a\mid b$,知有 $k\in\mathbf{N}$,使得 $b=ka$。于是:

$x\equiv y \pmod b$

$\Rightarrow \exists k'\in\mathbf{N}$,使得 $x-y=k'b$

$\Rightarrow x-y=k'ka=(k'k)a$

$\Rightarrow x\equiv y \pmod a$

3. 应用

整数关于模 n 的等价类有广泛的实际背景。例如,大写字母的编码值就构成一个模为 26 的等价类 $Z_{26}=\{0, 1, \cdots, 25\}$。具体地说,任何一个大写字母的编码值 x 减去'A'的编码值,必为 0~25 之一。下列的问题就涉及该等价类。

The Hardest Problem Ever

Description

Julius Caesar lived in a time of danger and intrigue. The hardest situation Caesar ever faced was keeping himself alive. In order for him to survive, he decided to create one of

the first ciphers. This cipher was so incredibly sound, that no one could figure it out without knowing how it worked.

You are a sub captain of Caesar's army. It is your job to decipher the messages sent by Caesar and provide to your general. The code is simple. For each letter in a plaintext message, you shift it five places to the right to create the secure message (i. e. , if the letter is 'A', the cipher text would be 'F'). Since you are creating plain text out of Caesar's messages, you will do the opposite:

Cipher text

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Plain text

V W X Y Z A B C D E F G H I J K L M N O P Q R S T U

Only letters are shifted in this cipher. Any non-alphabetical character should remain the same, and all alphabetical characters will be upper case.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be no blank lines separating data sets. All characters will be uppercase.

A single data set has 3 components.

(1) Start line - A single line, "START".

(2) Cipher message - A single line containing from one to two hundred characters, inclusive, comprising a single message from Caesar.

(3) End line - A single line, "END".

Following the final data set will be a single line, "ENDOFINPUT".

Output

For each data set, there will be exactly one line of output. This is the original message by Caesar.

Sample Input

```
START
NS BFW, JAJSYX TK NRUTWYFSHJ FWJ YMJ WJXZQY TK YWNANFQ HFZXJX
END
START
N BTZQI WFYMJW GJ KNWXY NS F QNYYQJ NGJWNFS ANQQFLJ YMFS XJHTSI NS WTRJ
END
START
IFSLJW PSTBX KZQQ BJQQ YMFY HFJXFW NX RTWJ IFSLJWTZX YMFS MJ
END
ENDOFINPUT
```

Sample Output

```
IN WAR, EVENTS OF IMPORTANCE ARE THE RESULT OF TRIVIAL CAUSES
```

I WOULD RATHER BE FIRST IN A LITTLE IBERIAN VILLAGE THAN SECOND IN ROME
DANGER KNOWS FULL WELL THAT CAESAR IS MORE DANGEROUS THAN HE

凯撒的部队间传递的消息加密规则很简单:消息由大写字母及标点符号组成。字母按各自与其后第 5 个字母对应,标点符号保持不变的方式转换成密文。你的任务是将收到的密文解密为明文。

为使行文简洁,将字符'A'、'B'、…、'Z'的编码值就用自身表示。设密文文本存储为数组 $cipher[1..n]$,$cipher[i](1\leqslant i\leqslant n)$为第 i 个字符的编码值。若 $cipher[i]$为一个大写字母,$cipher[i]$对应的明文字母的编码值为 x,则

$$(x-\text{'A'}+5) \bmod 26=cipher[i]-\text{'A'}$$

此表达式蕴涵着:

$$(cipher[i]-\text{'A'}-5) \bmod 26=x-\text{'A'}$$

即

$$x=((cipher[i]-\text{'A'}-5) \bmod 26)+\text{'A'}$$

利用此式,将解密过程实现为如下函数。

```
1 char *decode(char *cipher){
2   int n=strlen(cipher),i;
3   char *plain=(char*)calloc(n+1, sizeof(char));
4   for(i=0;i<n;i++)
5     if(cipher[i]<='Z'&&cipher[i]>='A')          /*cipher[i]是字母*/
6         plain[i]=(cipher[i]-'A'+21)%26+'A';  /*21与-5关于模26等价*/
7       else
8         plain[i]=cipher[i];
9   return plain;
10 }
```

程序 6-7 解决 The Hardest Problem Ever 问题的解密函数

对程序 6-7 的说明如下。

(1) 函数 decode 的参数 cipher 是表示消息密文的字符串。函数返回对 cipher 解密后的明文字符串。

(2) 函数体中定义的字符指针变量 plain 用来指向存储消息明文的字符串。pain 也是函数的返回值。

(3) 第 4～8 行的 **for** 循环对密文 cipher 中的每个字符检测是否为字母。第 6 行处理密文字母 cipher[i]。需要注意的是 C 语言中整数的求余运算 a%n 与数论中的模运算 a mod m 有一个微妙的区别:若 a 是负数,a%n 是一个负数。而 a mod n=a%n+n,则 a 是一个正数。例如,$-5 \bmod 26=-5\%26+26=21$。于是,解密表达式

$$((cipher[i]-\text{'A'}-5) \bmod 26)+\text{'A'}$$

在 C 语言中表示为第 6 行中的

(cipher[i]- 'A '+21)%26+ 'A '.

调用函数 decode 解决 The Hardest Problem Ever 问题的完整程序存储在文件夹 chap06\The Hardest Problem Ever 中的源文件 TheHardestProblemEver.c 中，读者可打开文件研读。

4. 素数与合数

如果一个整数($a>1$)只有平凡约数 1 和 a，则称其为是一个**素数**。素数有很多特性并在数论中扮演着核心角色。前 20 个素数依次为

2，3，5，7，11，13，17，19，23，29，31，37，41，43，47，53，59，61，67，71

有无穷多个素数，即由所有素数构成的集合 $\boldsymbol{P}$ 是无限集合。这是因为，若假定 $\boldsymbol{P}=\{p_1, p_2, \cdots, p_n\}$ 为一有限集合，考虑整数 $p= p_1p_2\cdots p_n+1$。若 p 有一个因数 k，则由于 $p_1, p_2, \cdots, p_n$ 为素数，所以 k 不是任何一个 $p_i(1\leqslant i\leqslant n)$ 的因数，因此也不是积 $p_1p_2\cdots p_n$ 的因数。这样将得到 $k|1$ 的矛盾。这说明 p 没有因数。这又与所有素数构成的集合 $\boldsymbol{P}=\{p_1, p_2, \cdots, p_n\}$ 为一有限集合矛盾，所以 $\boldsymbol{P}$ 是无限集合。一个整数 $a>1$ 不是素数则称其为**合数**。例如，39 是合数，因为 $3 \mid 39$。整数 1 称为单位，它既不是素数也不是合数。类似地，整数 0 和所有的负整数都既不是素数也不是合数。

5. 公约数和最大公约数

若整数 d 是 a 的约数且 d 也是 b 的约数，则 d 是 a 和 b 的**公约数**。例如，30 的约数是 1、2、3、5、6、10、15 和 30，因此，24 与 30 的公约数是 1、2、3 和 6。注意 1 是任意两个整数的公约数。

公约数的一个重要性质如下：

$$d \mid a \text{ 且 } d \mid b \text{ 蕴含着 } d \mid (a+b) \text{ 及 } d \mid (a-b)$$

更一般地，对任意的整数 x 和 y，有

$$d \mid a \text{ 且 } d \mid b \text{ 蕴含着 } d \mid (ax+by) \tag{6-9}$$

此外，若 $a \mid b$，则或 $|a|\leqslant|b|$ 或 $b=0$，这意味着：

$$a \mid b \text{ 且 } b \mid a \text{ 蕴含着 } a=\pm b$$

两个不全为 0 的整数 a 和 b 的最大公约数是 a 和 b 的公约数的最大者。记为 $\gcd(a, b)$。例如，$\gcd(24, 30)=6$，$\gcd(5, 7)=1$，以及 $\gcd(0, 9)=9$。若 a 和 b 不全为 0，则 $\gcd(a, b)$ 是介于 1 与 $\min(|a|, |b|)$ 之间的一个整数。约定 $\gcd(0, 0)$ 为 0；这一约定对于建立 gcd 函数(如式(6-10))的标准通用性是必需的。

gcd 函数的基本性质如下。

$$\gcd(a, b) = \gcd(b, a) \tag{6-10}$$

$$\gcd(a, b) = \gcd(-a, b) \tag{6-11}$$

$$\gcd(a, b) = \gcd(|a|, |b|) \tag{6-12}$$

$$\gcd(a, 0) = |a| \tag{6-13}$$

$$\text{对任意的 } k \in \mathbf{Z}, \gcd(a, ka) = |a| \tag{6-14}$$

下列定理给出了 $\gcd(a, b)$ 的另一个特征。

定理 6-3　若 a 和 b 是任意不全为 0 的整数，则 $\gcd(a, b)$ 是 a 和 b 的线性组合的集合

$\{ax+by: x, y\in \mathbf{Z}\}$中正的元素的最小者。

证明 设 s 是 a 和 b 的最小的正的线性组合,并设 $s=ax+by$,其中 $x, y\in \mathbf{Z}$。设 $q=\lfloor a/s \rfloor$。则:

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax+by) \\ &= a(1-qx)+b(-qy) \end{aligned}$$

所以 $a \bmod s$ 也是 a 及 b 的线性组合。由于 $a \bmod s<s$,有 $a \bmod s=0$,这是因为 s 是这样的线性组合中正的最小者。所以 $s \mid a$,并且由类似的理由,$s \mid b$。于是,s 是 a 和 b 的一个公约数,所以 $\gcd(a, b)\geqslant s$。式(6-9)蕴含着 $\gcd(a, b) \mid s$,这是因为 $\gcd(a, b)$既能整除 a 又能整除 b,且 s 是 a 与 b 的线性组合。但 $\gcd(a, b) \mid s$ 且 $s>0$ 意味着 $\gcd(a, b)\leqslant s$。结合 $\gcd(a, b)\geqslant s$ 和 $\gcd(a, b)\leqslant s$ 推导出 $\gcd(a, b)=s$,即 s 是 a 和 b 的最大公约数。

推论 6-1 对任意的整数 a 和 b,若 $d \mid a$ 且 $d \mid b$,则 $d \mid \gcd(a, b)$。

证明 这是因为按定理 6-3,$\gcd(a, b)$是 a 和 b 的线性组合。再利用式(6-9),此推论得证。

推论 6-2 对所有的整数 a 和 b 以及任意非负整数 n:

$$\gcd(an, bn) = n \gcd(a, b)$$

证明 若 $n=0$,推论不言而喻。若 $n>0$,则 $\gcd(an, bn)$是集合$\{anx+bny\}$中正的最小者,而它是集合$\{ax+by\}$中正的最小者的 n 倍。

推论 6-3 对所有的正整数 n、a 和 b,若 $n \mid ab$ 且 $\gcd(a, n)=1$,则 $n \mid b$。

证明 由于 $\gcd(a, n)=1$,根据定理 6-3,有整数 x 和 y,使得 $ax+ny=1$。于是,$abx+nby=b$。等式的左边两项均能被 n 整除,这蕴含着右边的 b 能被 n 整除。

6. 互质

两个整数 a、b,若它们只有公约数 1,即若 $\gcd(a, b)=1$,则称它们是**互质**的。例如,8 和 15 是互质的,这是因为 8 的约数是 1、2、4 和 8,而 15 的约数是 1、3、5 和 15。定理 6-4 断言若两个整数都与整数 p 互质,则它们的乘积也与 p 互质。

定理 6-4 对任意的整数 a、b 和 p,若有 $\gcd(a, p)=1$ 且 $\gcd(b, p)=1$,则 $\gcd(ab, p)=1$。

证明 由定理 6-3,存在整数 x、y、x'和 y'使得

$$\begin{aligned} ax + py &= 1 \\ bx' + py' &= 1 \end{aligned}$$

将此两个等式相乘,并整理得:

$$ab(xx') + p(ybx' + y'ax + pyy') = 1$$

由于 1 是 ab 和 p 的正的线性组合,根据定理 6-3 就完成了证明。

说整数 n_1、n_2、…、n_k 是两两互质的,若只要 $i\neq j$,就有 $\gcd(n_i, n_j)=1$。

定理 6-5 对任意非 0 整数 a 和 b,若 $d=\gcd(a, b)$,则 $\gcd(a/d, b/d)=1$。

证明 设 $c=\gcd(a/d, b/d)$,则 $c\geqslant 1$。为证明 $c=1$,只需证明 $c\leqslant 1$。由于 $c|(a/d)$且 $c|(b/d)$,所以 $cd|a$ 且 $cd|b$。根据推论 6-1 知 $cd|d$,这意味着 $cd\leqslant d$。由于 $d\geqslant 1$,故 $c\leqslant 1$。由此可见,$1=c=\gcd(a/d, b/d)$。

推论 6-4　若 $ac \equiv bc \pmod n$，$\gcd(c, n)=d$，则 $a \equiv b \pmod{n/d}$。

证明　由 $ac \equiv bc \pmod n$，得 $n \mid c(a-b)$，即 $(n/d) \mid ((c/d)(a-b))$。按定理 6-5 知 $\gcd(c/d, n/d)=1$，由推论 6-3 知 $(n/d) \mid (a-b)$。此即 $a \equiv b \pmod{n/d}$。

7. 唯一分解

关于被素数整除的一个基本却很重要的事实如下。

定理 6-6　对所有的素数 p 和所有的整数 a、b，若 $p \mid ab$，则 $p \mid a$ 或 $p \mid b$（或两者都有）。

证明　用反证法。假定 $p \mid ab$ 但 p　a 且 p　b。于是，$\gcd(a, p)=1$ 且 $\gcd(b, p)=1$，这是因为 p 的约数是 1 和 p，且由假设 p 既不能整除 a 也不能整除 b。则定理 6-4 蕴含着 $\gcd(ab, p)=1$，此与 $p \mid ab$ 且由此而得的 $\gcd(ab, p)=p$ 矛盾。这一矛盾就完成了证明。

定理 6-6 的一个推论是一个整数有唯一的素因数分解。

定理 6-7（唯一分解）　合数 a 可以写成唯一的乘积形式 $a=p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$，其中 p_i 是素数，$p_1<p_2<\cdots<p_r$，e_i 是正整数。

证明　设正整数 $a=p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}=q_1^{c_1} q_2^{c_2} \cdots q_t^{c_t}$，其中 p_i、q_j（$i=1, 2, \cdots, r$，$j=1, 2, \cdots, t$）为素数，且 $p_1<p_2<\cdots<p_r$ 及 $q_1<q_2<\cdots<q_t$。显然，$\forall i, 1 \leqslant i \leqslant t$，应有 $q_i^{c_i} \mid p_1^{e_1} \cdots p_r^{e_r}$，根据定理 6-6：

$$\exists |^{①} j, 1 \leqslant j \leqslant r, \text{使得 } q_i^{c_i} \mid p_j^{e_j} \Rightarrow q_i = p_j \text{ 且 } c_i \leqslant e_j \tag{6-15}$$

相仿地可得 $\forall j, 1 \leqslant j \leqslant r$：

$$\exists | \, i, 1 \leqslant i \leqslant t, p_j = q_i \text{ 且 } e_j \leqslant c_i \tag{6-16}$$

由式(6-15)和式(6-16)表示的对应的唯一性，有 $t=r$，$\forall 1 \leqslant j \leqslant r$，必有 $p_j = q_j$ 且 $c_j = e_j$。定理由此得证。

作为例子，数 6000 可以被唯一地分解为 $2^4 \times 3 \times 5^3$。

6.3　最大公约数

本节中，描述有效计算两个整数的最大公约数的 Euclid 算法。运行时间的分析涉及 Fibonacci 数，由此导出 Euclid 算法的最坏情形的输入。

本节中，限于讨论非负整数。这一限制根据式(6-12)是正确的，该式断言 $\gcd(a, b)=\gcd(|a|, |b|)$。

理论上，可以由正整数 a 和 b 的素因数分解来计算 a 和 b 的 $\gcd(a, b)$。事实上，若：

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \tag{6-17}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} \tag{6-18}$$

则：

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} \tag{6-19}$$

然而，如将在 6.6 节中看到的那样，到现在为止，即使是最好的因数分解算法都不是多项式时间的，所以这个计算最大公约数的方案并不是一个有效的算法。

① ∃| 表示“存在唯一的……”。

快速计算最大公约数的 Euclid 算法基于以下定理。

定理 6-8(GCD 递归定理) 对任一非负整数 a 和任一正整数 b,$\gcd(a, b)=\gcd(b, a \bmod b)$。

证明 我们将证明 $\gcd(a, b)$和 $\gcd(b, a \bmod b)$互相整除,所以根据式(6-9)它们必相等(这是因为它们都是非负的)。若设 $d=\gcd(a, b)$,则 $d \mid a$ 和 $d \mid b$。根据定理 6-1,$(a \bmod b)=a-qb$,其中 $q=\lfloor a/n \rfloor$。这样,$(a \bmod b)$可视为 a 和 b 的线性组合,式(6-9)蕴含着 $d \mid (a \bmod b)$。于是,由于 $d \mid b$ 和 $d \mid (a \bmod b)$,根据推论 6-1,$d \mid \gcd(b, a \bmod b)$或等价地

$$\gcd(a, b) \mid \gcd(b, a \bmod b) \tag{6-20}$$

证明 $\gcd(b, a \bmod b) \mid \gcd(a, b)$几乎是一样的。今设 $d=\gcd(b, a \bmod b)$,则 $d \mid b$ 且 $d \mid (a \bmod b)$。由于 $a=qb+(a \bmod b)$,其中 $q=\lfloor a/b \rfloor$,有 a 是 b 和$(a \bmod b)$的线性组合。根据式(6-9),推出 $d \mid a$。由于 $d \mid b$ 且 $d \mid a$,根据推论 6-1 有 $d \mid \gcd(a, b)$或等价地

$$\gcd(b, a \bmod b) \mid \gcd(a, b) \tag{6-21}$$

6.3.1 Euclid 算法

Euclid(约公元前 300 年)原理描述了下列 gcd 算法,尽管它可能源于更早。Euclid 算法表示成直接基于定理 6-8 的一个递归过程。输入的 a 和 b 是任意的非负整数。

```
EUCLID(a, b)
1  if b=0
2      then return a
3      else return EUCLID(b, a mod b)
```

算法 6-6 计算 B 进制正整数 a、b 的最大公约数的递归过程

作为运行 EUCLID 的一个例子,考虑计算 gcd(30, 21):

$$\begin{aligned}\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9)\\ &= \text{EUCLID}(9, 3)\\ &= \text{EUCLID}(3, 0)\\ &= 3\end{aligned}$$

在此计算中,三次调用 EUCLID。

EUCLID 的正确性源于定理 6-8 和如下事实:若算法在第 2 行返回 a,则 $b=0$,故式(6-13)蕴含着 $\gcd(a, b)=\gcd(a, 0)=a$。算法不会无限递归,这是因为每次递归调用的第二个参数都严格递减且总是非负的。所以,EUCLID 总是正确终止。

6.3.2 EUCLID 算法的运行时间

把 EUCLID 的运行时间作为 a 和 b 的比特位数的函数来加以分析。不失一般性,假定 $a>b\geqslant 0$。这一假定是正确的,因为若 $b>a\geqslant 0$,则 EUCLID(a, b)马上就递归地调用 EUCLID(b, a)。也就是说,如果第一个参数比第二个参数小,EUCLID 将花费一次递归调用来交换

它们，然后再来处理。类似地，若 $b=a>0$，过程在一次递归调用后终止，因为 $a \bmod b=0$。

EUCLID 总的运行时间正比于它所进行的递归调用的次数。我们的分析要用到下列定义的 **Fibonacci 数** F_k。

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \quad i \geqslant 2 \end{aligned}$$

于是，每个 Fibonacci 数都是它前面两个数的和。这就得出了序列：

0，1，1，2，3，5，8，13，21，34，55，…

引理 6-3　若 $a>b\geqslant 1$ 且 EUCLID(a, b)执行了 $k\geqslant 1$ 次递归调用，则 $a\geqslant F_{k+2}$ 且 $b\geqslant F_{k+1}$。

证明　对 k 做数学归纳。对 $k=1$ 时，$b\geqslant 1=F_2$，且由于 $a>b$，必有 $a\geqslant 2=F_3$。由于 $b>(a \bmod b)$，在每次递归调用中第一个参数严格大于第二个参数；$a>b$ 的假定对每个递归调用都是成立的。

归纳地假定引理对 $k-1$ 次递归调用是真的；将证明对 k 次递归调用也是真的。由于 $k>0$，有 $b>0$ 且 EUCLID(a, b)递归地调用 EUCLID$(b, a \bmod b)$，而 EUCLID$(b, a \bmod b)$要做 $k-1$ 次递归调用。根据递归假设，有 $b\geqslant F_{k+1}$，且$(a \bmod b)\geqslant F_k$。按本引理的条件 $a>b>0$，此蕴含着$\lfloor a/b \rfloor\geqslant 1$。于是，

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor b) \\ &\leqslant a \end{aligned}$$

因此，

$$\begin{aligned} a &\geqslant b + (a \bmod b) \\ &\geqslant F_{k+1} + F_k \\ &= F_{k+2} \end{aligned}$$

下列定理是此引理的直接推论。

定理 6-9(Lamé 定理)　对任一整数 $k\geqslant 1$，若 $a>b\geqslant 1$ 且 $b<F_{k+1}$，则 EUCLID(a, b)执行的递归调用次数少于 k。

可以证明定理 6-9 的上界是最好的。连续的 Fibonacci 数是 EUCLID 的最坏情形的输入。由于 EUCLID(F_3, F_2)恰做了一次递归调用，且因为对 $k>2$，有 $F_{k+1} \bmod F_k = F_{k-1}$，所以

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, (F_{k+1} \bmod F_k)) \\ &= \gcd(F_k, F_{k-1}) \end{aligned}$$

于是，EUCLID(F_{k+1}, F_k)恰递归 $k-1$ 次，符合定理 6-9 的上界。

Fibonacci 数列与如下定义的黄金比率 φ 及其共轭有关：

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots, \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2} = -0.61803\cdots$$

明确地说，可以用数学归纳法证明：

$$F_k = \frac{\varphi^k - \hat{\varphi}^k}{\sqrt{5}}$$

由于$|\hat{\varphi}|<1$，有$|\hat{\varphi}^i|/\sqrt{5}<1/\sqrt{5}<1/2$，所以第 k 个 Fibonacci 数等于 $\varphi^k/\sqrt{5}$ 的四舍五入后的

整数值。所以,EUCLID 的递归调用次数为 $O(\lg b)$。这就推导出若将 EUCLID 用于两个 β 二进制位的整数它将执行 $O(\beta)$次算术运算和 $O(\beta^3)$次位运算(假定 β 二进制位整数的乘法和除法执行 $O(\beta^2)$次位运算)。

6.3.3 Euclid 算法的迭代版本

递归过程 EUCLID 是末尾递归,很容易写出与之等价的迭代版本。

```
GCD(a, b)
while b≠0
    do r←a mod b
       a←b
       b←r
return a
```

算法 6-7 用辗转相除法计算 B 进制整数 a、b 的最大公约数的过程

数论把上述的 GCD 过程称为辗转相除法。

若 $b=0$,则 $\gcd(a, b)=a$。否则

$q_0 \leftarrow \lfloor a/b \rfloor, r_0 \leftarrow a \bmod b(\neq 0), a = q_0 b + r_0$;

$q_1 \leftarrow \lfloor b/r_0 \rfloor, r_1 \leftarrow b \bmod r_0(\neq 0), b = q_1 r_0 + r_1$;

$\vdots$

$q_i \leftarrow \lfloor r_{i-2}/r_{i-1} \rfloor, r_i \leftarrow r_{i-2} \bmod r_{i-1}(\neq 0), r_{i-2} = q_i r_{i-1} + r_i$

$\vdots$

$q_{m-1} \leftarrow \lfloor r_{m-3}/r_{m-2} \rfloor, r_{m-1} \leftarrow r_{m-3} \bmod r_{m-2}(\neq 0), r_{m-3} = q_{m-1} r_{m-2} + r_{m-1}$

$q_m \leftarrow \lfloor r_{m-2}/r_{m-1} \rfloor, r_m \leftarrow r_{m-2} \bmod r_{m-1}(=0), r_{m-2} = q_m r_{m-1}$

根据定理 6-7,$r_{m-1}=\gcd(r_{m-2}, r_{m-1})= \gcd(r_{m-3}, r_{m-2})=\cdots= \gcd(r_{i-1}, r_i)=\cdots \gcd(r_0, r_1)=\gcd(b, r_0)=\gcd(a, b)$。

由第 1 个式子得

$$r_0 = a - q_0 b$$

由第 2 个式子得

$$\begin{aligned} r_1 &= b - q_1 r_0 \\ &= (-q_1)a + (q_0 + 1)b \\ &\cdots \end{aligned}$$

一般地,对 $0 \leqslant j < i$,r_j 可表为 $r_j = x_j a + y_j b$,则

$$\begin{aligned} r_i &= r_{i-2} - q_i r_{i-1} \\ &= (x_{i-2}a + y_{i-2}b) - q_i(x_{i-1}a + y_{i-1}b) \\ &= (x_{i-2} - q_i x_{i-1})a + (y_{i-2} - q_i y_{i-1})b \end{aligned}$$

于是,若设

$$x_{-2} = 1, \quad y_{-2} = 0$$
$$x_{-1} = 0, \quad y_{-1} = 1$$

则可得递推公式：

$$x_i = x_{i-2} - q_i x_{i-1},\quad y_i = y_{i-2} - q_i y_{i-1},\quad i = 0,1,2,\cdots$$

利用此递推公式就可在用辗转相除法计算 a、b 最大公余数 d 的同时，计算出满足定理 6-3 表示的线性组合系数 x 与 y，使得 $d=\gcd(a,b)=xa+yb$。写成伪代码过程如下。

```
EXTENDED-GCD(a, b)
x1←0, x←1, y1←1, y←0
while b≠0
  do q←⌊a/b⌋, r←a mod b, x2← x −qx1, y2← y −qy1
     a←b, b←r, x←x1, x1←x2, y←y1, y1←y2
return (a, x, y)
```

算法 6-8　计算定理 6-3 描述的最大公约数线性组合的过程

算法 6-8 与算法 6-7 一样，是一个循环结构。并且循环条件以及循环体内对 a、b 及 r 的处理也是一样的。所以，算法 6-8 的运行时间也是 $O(\beta^3)$，其中 β 表示 a、b 的位数。

6.3.4　程序实现

本章所有算法都可以对系统定长整型数据和在前定义的基数为 Base$=10^9$ 的长整数 BigInt 数据加以实现。本节就计算非负整数 a、b 的最大公约数为例，列出其对两种类型数据的实现代码，并加以详细解析。读者可对比深入领会数论算法的特点——对位的操作。

```
typedef unsigned long long ul_int;
typedef long long l_int;
ul_int egcd(ul_int a, ul_int b, l_int *x, l_int *y){
 ul_int q, r;
 l_int x1=0, x2, y1=1, y2;
 *x=1; *y=0;
 while(b){
   q=a/b;r=a%b;x2=*x-q*x1;y2=*y-q*y1;
   a=b;b=r;*x=x1;x1=x2;*y=y1;y1=y2;
 }
  return a;
}
BigInt extendedEuclid(BigInt a, BigInt b, BigInt* x, BigInt* y){
  BigInt r={NULL,0},q={NULL,0},t1={NULL,0},t2={NULL,0},
         x1=newIntBystring("0"),                /* x1←0 */
         y1=newIntBystring("1");                /* y1←1 */
         x2={NULL,0}, y2={NULL,0},a1={NULL,0},b1={NULL,0},
         intAssign(&a1,a),intAssign(&b1,b),     /* a1←a, b1←b */
         intAssign(x, y1),intAssign(y, x1);     /* x←y1,y←x1 */
  a1.sign=b1.sign=0;
  while(!isZero(b1)){                           /* while b≠0 */
    clrBigInt(&q),clrBigInt(&r);                /* 清理 q、r 空间 */
```

```
23     r.value=createList(sizeof(long),NULL);
24     q.value=valueDivision(a1.value,b1.value, r.value);/* q←a/b, r←a mod b */
25     clrBigInt(&t1),clrBigInt(&t2),t1=product(q, x1),t2=product(q, y1);
26     clrBigInt(&x2),clrBigInt(&y2),                /* 清理 x2、y2 的空间 */
27     x2=diff(*x,t1), y2=diff(*y,t2);               /* x2←x-qx1, y2←x-qy2 */
28     intAssign(&a1,b1);intAssign(&b1,r);           /* a←b, b←r */
29     intAssign(x, x1);intAssign(y,y1);             /* x←x1, y←y1 */
30     intAssign(&x1,x2);intAssign(&y1,y2);          /* x1←x2,y1←y2 */
31   }
32   clrBigInt(&b1);clrBigInt(&q);clrBigInt(&r);clrBigInt(&x1);/* 清理各临时变量的空间 */
33   clrBigInt(&y1);clrBigInt(&x2);clrBigInt(&y2);clrBigInt(&t1);clrBigInt(&t2);
34   return a1;                                      /* return a */
35 }
```

程序 6-8　实现算法 6-8 EXTENDED-GCD 过程的 C 函数

对程序 6-8 的说明如下。

(1) 为使代码简洁,第 1 行和第 2 行定义 **unsingned long long** 和 **long long** 的别名分别为 ul_int 和 l_int。

(2) 第 3～12 行定义的函数 egcd 实现算法 6-8 的 EXTENDED-GCD 过程,对参数 a、b 表示的两个 ul_int 型数据计算最大公约数 d,以及 d 关于 a、b 线性组合式中的系数 x 和 y。函数的返回值为 a、b 的最大公约数 d,而 d 关于 a、b 线性组合的系数由指针参数 x、y 指引。函数体内的程序代码与算法过程的伪代码十分接近。

(3) 第 13～35 行定义的函数 extendedEuclid 是实现算法 6-8 的 BigInt 版本。函数返回参数 a、b 的最大公约数 d,d 关于 a、b 的线性组合系数由指针参数 x、y 指引。

函数体中定义的变量 x1、x2、y1、y2 与算法中同名变量的意义相同。

在 a、b 最大公约数的计算过程中要改变 a、b 的值,虽然 C 函数的参数是按值传递,但由于 a、b 是 BigInt 类型的对象,其成员属性 value 是指向链表的指针,改变 a、b 的值就会造成 value 指引的链表中的结点数据发生变化,所以设置变量 a1、b1 作为 a、b 在运算过程中的替身。

由于 BigInt 类型对象的 value 属性是一个链表指针,对其直接赋值 value 将指向另一个链表执行赋值运算 a=b 后的内存情形如图 6-1(a)所示,所以,为向一个 BigInt 对象 a 赋值 b,需要先对对象 a 做清空 value 空间的操作,清空 BigInt 对象 value 属性空间的操作由函数 clrBigInt 完成,其原型声明为

```
void clrBigInt(BigInt * a);
```

除了对一个 BigInt 变量赋值前需要执行 clrBigInt 操作外,程序执行完毕前,所有临时的 BigInt 变量也要执行此操作,以防内存泄漏。这样可能导致数据操作的安全性,然后将 b 的 value 链表复制给 a 的 value。

将一个 BigInt 对象 a 克隆为 b 的操作由函数 intAssign 完成,调用函数 intAssign(&a,b)的内存情形如图 6-1(b)所示。其原型声明为

```
void intAssign(BigInt *a, BigInt a);
```

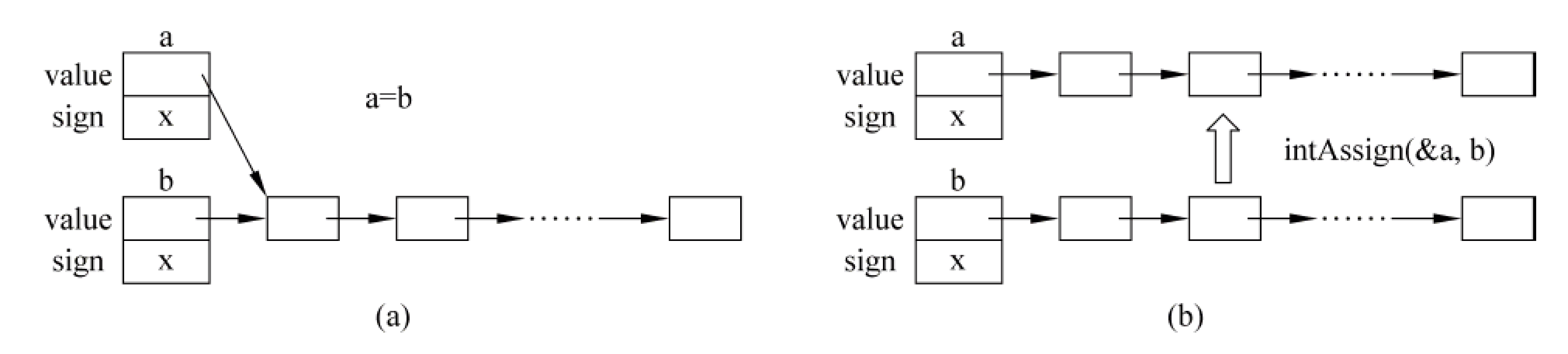


图 6-1　BigInt 变量的赋值与复制

在过程运行时，需要做迭代 $x_2 \leftarrow x - qx_1$，$y_2 \leftarrow x - qy_2$，在本函数中 x、y、x1、y1、x2、y2、q 都是 BigInt 型对象，它们之间的算术运算都需要调用事先定义好的函数 product 和 diff 来完成，函数返回的也是 BigInt 对象，直接将函数值赋值，则下一次迭代时就可能丢弃原来的 value 属性，那将造成内存泄漏。因此，设置中间变量 t1、t2 接受运算函数 product 调用返回值，每次迭代前先调用函数 clrBigInt 清空原来的空间。

计算两个 BigInt 对象 a 与 b 的积 ab 的函数 product 的原型声明如下：

```
BigInt product(BigInt a, BigInt b);
```

计算两个 BigInt 对象 a 与 b 的差 a－b 的函数 diff 的原型声明如下：

```
BigInt diff(BigInt a, BigInt b);
```

上述 4 个函数均定义在 numbertheory 文件夹中的源文件 bigint. c 内，它们的原型声明于同一文件夹内的头文件 bigint. h 中。

计算 a、b 的绝对值相除的商和余数用一个 valueDivision 函数完成，其原型声明如下：

```
LinkedList * valueDivision(LinkeList * x, LinkedList * y, LinkedList * r);
```

该函数计算存储于 x、y 中的两个大整数的绝对值相除的商和余数，返回商，余数由指针参数 r 指引。函数定义于 numbertheory 文件夹中的源文件 valuecalc. c 中，其原型声明于同一文件夹内的头文件 valuecalc. h 中。

程序 6-5 中定义的函数存储于 numbertheory 文件夹中的源文件 gcd. c 中，原型声明于同一文件夹的头文件 gcd. h 中。在这两个文件中还存储有实现算法 6-7 的 C 函数的 ul_int 版本和 BigInt 版本代码，读者可打开文件研读。

6. 3. 5　应用

Jugs

Description

In the movie "Die Hard 3", Bruce Willis and Samuel L. Jackson were confronted with the following puzzle. They were given a 3-gallon jug and a 5-gallon jug and were asked to fill the 5-gallon jug with exactly 4 gallons. This problem generalizes that puzzle.

You have two jugs, A and B, and an infinite supply of water. There are three types of actions that you can use: (1) you can fill a jug, (2) you can empty a jug, and (3) you can

pour from one jug to the other. Pouring from one jug to the other stops when the first jug is empty or the second jug is full, whichever comes first. For example, if A has 5 gallons and B has 6 gallons and a capacity of 8, then pouring from A to B leaves B full and 3 gallons in A.

A problem is given by a triple (a, b, n), where a and b are the capacities of the jugs A and B, respectively, and n is the goal. A solution is a sequence of steps that leaves exactly n gallons in jug B. The possible steps are

fill A
fill B
empty A
empty B
pour A B
pour B A
success

where "pour A B" means "pour the contents of jug A into jug B", and "success" means that the goal has been accomplished.

You may assume that the input you are given does have a solution.

Input

Input to your program consists of a series of input lines each defining one puzzle. Input for each puzzle is a single line of three positive integers: a, b, and n. a and b are the capacities of jugs A and B, and n is the goal. You can assume $0 < a \leqslant b$ and $n \leqslant b \leqslant 1000$ and that A and B are relatively prime to one another.

Output

Output from your program will consist of a series of instructions from the list of the potential output lines which will result in either of the jugs containing exactly n gallons of water. The last line of output for each puzzle should be the line "success". Output lines start in column 1 and there should be no empty lines nor any trailing spaces.

Sample Input

```
3 5 4
5 7 3
```

Sample Output

```
fill B
pour B A
empty A
pour B A
fill B
pour B A
success
fill A
```

pour A B
fill A
pour A B
empty B
pour A B
success

1. 问题分析与算法描述

两个容量分别为 a 和 b 加仑的水桶($\gcd(a, b)=1$ 且 $a<b$)A 和 B,并且有无限量的水,通过若干次灌满某个桶、将某个桶中的水倒掉、将某个桶中的水倒进另一个桶中,这样的操作,使得 B 桶中刚好有 $n(<b)$加仑的水。任务是编写程序对给定的 a、b、n,列出从 A、B 为空桶开始到达成目标(B 桶中刚好有 n 加仑水)为止的操作序列。

为解决此问题,要做两步工作:首先要确定从水池中要对某桶注满多少次,倒掉另一桶多少次能使得 B 桶中的水恰有 n 加仑,即确定整数 x、y 使得

$$ax + by = n$$

其中 x、y 必有一正一负,正者表示要注满的桶数,负者表示要倒掉的桶数。该方程在数论中称为丢番图方程。该方程有解的充分必要条件是 $\gcd(a, b) \mid n$。本问题中,由于 $\gcd(a, b)=1$,故必有解。若 x_0、y_0 为上述丢番图方程的一个解,则

$$\begin{cases} x = x_0 + bt \\ y = y_0 - at \end{cases} \quad t \in \mathbf{Z}$$

为其所有解。也就是说该方程有无穷多个解。我们的目标是求出使得倒掉的水最少的 x 和 y。写成伪代码过程如下。

GET-XY(a, b, n)
1 $(d, x, y) \leftarrow$ EXTENDED-GCD(a, b)
2 $x_0 \leftarrow xn$, $y_0 \leftarrow yn$
3 $min \leftarrow \infty$
4 $t \leftarrow 0$, $i \leftarrow 1$, $x_1 \leftarrow x_0$, $y_1 \leftarrow y_0$
5 $x_2 \leftarrow x_1 + bi$, $y_2 \leftarrow y_1 - ai$
6 **while** $|x_2 a| \leqslant |x_1 a|$ or $|y_2 b| \leqslant |y_1 b|$
7 　**do if** $min >$ $x_2 a$, $y_2 b$ 中负数的绝对值
8 　　**then** $min \leftarrow$ $x_2 a$, $y_2 b$ 中为负数的绝对值
9 　　　$t \leftarrow i$
10 　　　$x_1 \leftarrow x_2$, $y_1 \leftarrow y_2$
11 　　　$i \leftarrow i+1$
12 　　　$x_2 \leftarrow x_1 + bi$, $y_2 \leftarrow y_1 - ai$
13 **return** $x_0 + bt$, $y_0 - at$

算法 6-9　计算使得倒掉的水最少的 $ax+by=n$ 的解 x、y

该过程的运行如下。第 1 行计算 $\gcd(a, b)$的线性组合系数 x、y,即 $ax+by=1$。两边同乘 n 得到 $axn+byn=n$。第 2 行令 x_0、y_0 分别为 xn 和 yn,则 x_0、y_0 为 $ax+by=n$ 的一个解。第 6～12 行的 **while** 循环通过迭代计算使得倒掉的水最少的 x 和 y 的值。

应当指出,丢番图方程的一般解中,参数 t 是在整数集 $\mathbf{Z}$ 中取值。而在算法过程中,t 取正整数。第 6~12 行的循环条件是 xa 和 yb 不同时增长。因为一旦两者同时增长,就会越变越大。这样,上述 *GET-XY* 过程仅适合于 $x_0<0$ 的情形。对于 $y_0<0$ 的情形,迭代式应改成 $x_2=x_1-bt$, $y_2=y_1+at$。

其次,根据得到的 x 和 y,确定操作步骤。所有可能的步骤均为如下 3 种操作之一。

(1) 灌满 B 桶(或灌满 A 桶)。

(2) 倒掉 A 桶(或倒掉 B 桶)。

(3) 将 B 桶中的水倒入 A 桶(或将 A 桶中的水倒入 B 桶)。

在 x<0 条件下,即灌满 y 桶 B 倒掉 x 桶 A 就可得到 n 加仑水,伪代码过程如下。

```
JUGS (a, b, n, x, y)
1  a1←b1←0
2  while b1≠n
3    do if y>0 and b1=0
4         then b1←b, y←y-1                     ▷灌满 B 桶
5              print "fill B"
6              继续下一轮重复
7       if x<0 and a1=a
8          then a1←0, x←x+1                    ▷将 A 桶中的水倒掉
9               print "empty A"
10              继续下一轮重复
11      if 0<b1≤a-a1
12        then b1←0, a1←a1+b1                  ▷将 B 桶中的水全倒进 A 桶
13        else b1← b1-(a-a1), a1←a             ▷用 B 桶中的水灌满 A 桶
14      print "pore B A"
```

算法 6-10 解决 Jugs 问题的一个案例的算法过程

该过程运行如下。第 1 行将表示 A 桶和 B 桶中水量的变量 a_1 和 b_1 初始化为 0。第 2~14 行的 while 循环模拟一台自动机在①B 桶为空;②A 桶被灌满;③B 通可倒水到 A 桶 3 个状态之间的转换。其中,第 3~6 行的 **if** 结构将状态①转换为状态③。而第 7~10 行的 **if** 结构是将状态②转换为状态③。第 11~14 行将状态③转换为状态①,B 桶被倒空或状态② A 桶被倒满。循环直至 B 桶中的水量 b_1 为 n 为止。

2. 程序实现

实现算法过程 GET-XY 如下。

```
void getxy(int a, int b, int n, int *x, int *y){
  long long x0, y0, t=0, i=1, min=LONG_MAX,f ,x1, y1, x2, y2;
  egcd(a, b, &x0, &y0);                 /*计算 a、b 的 gcd 线性组合系数 x0、y0*/
  f=x0<0? 1:-1;                         /*确定符号因子*/
  x0*=n; y0*=n;                         /*计算丢番图方程的初始解*/
  x1=x0;y1=y0;                          /*计算迭代解初始值*/
  x2=x1+f*b*i;y2=y1-f*a*i;              /*计算首次迭代解*/
```

```
8   while(abs(x2)<abs(x1) || abs(y2)<abs(y1)){          /* 解的绝对值不同时增大 */
9       if(x2<0&&-x2 * a<min){
10          min=-x2 * a;
11          t=i;
12      }
13      if(y2<0&&-y2<min){
14          min=-y2 * b;
15          t=i;
16      }
17      x1=x2;y1=y2;
18      i++;
19      x2=x1+f * b * i;y2=y1+f * a * i;
20  }
21  * x=x0+f * b * t; * y=y0-f * a * t;
22 }
```

程序 6-9　实现算法 6-9 的 C 函数

对程序 6-8 的说明如下。

(1) 函数 getxy 有 5 个参数。参数 a、b、n 的意义与伪代码过程的 3 个同名参数一致，表示 A 桶、B 通的容量和要在 B 桶中保留的目标量。指针参数 x、y 用来向外部返回所求的丢番图方程的解。

(2) 函数 getxy 与伪代码过程不同，它不仅要解 x0<0 的情形，也要解 y0<0 的情形。这两种情形影响迭代表达式中 t 的符号。因此，设置符号因子 f，并在第 4 行对其进行了初始化，使得能在迭代过程中找到最小的解。

(3) 第 8～20 行的 **while** 循环实现算法过程中的第 6～12 行的迭代循环。其中，9～12 行和第 13～16 行的两个 **if** 语句跟踪当前监测到的使得倒掉的水最少的解对应的 t。第 21 行用此 t 计算最终得到的解。

对 JUGS 过程的实现代码如下。

```
char  * jugs(int a, int b, int n, int x, int y){
 int a1=0, b1=0, a2, b2,x1,y1, * bb;
 char A[2], B[2], * s=(char * )calloc(1000, sizeof(char));
 if(x<0){
     a2=a;b2=b;bb=&b1;
     strcpy(A,"A");strcpy(B,"B");
     x1=x;y1=y;
 }else{
     a2=b;b2=a;bb=&a1;
     strcpy(A,"B");strcpy(B,"A");
     x1=y;y1=x;
 }
 while( * bb!=n){
     if(y1&&b1==0){                       /* B 桶汲水 */
         b1=b2;y1--;
```

```
16          strcat(s, "fill ");strcat(s,B);strcat(s,"\n");
17          continue;
18      }
19      if(x1&&a1==a2){                    /*将A桶中的水倒掉*/
20          a1=0;x1++;
21          strcat(s,"empty ");strcat(s,A);strcat(s,"\n");
22          continue;
23      }
24      if(b1<=a2-a1){                     /*B桶中的水可全部倒入A桶*/
25          a1=b1+a1;b1=0;
26      }else{                             /*B桶中的水只能部分倒入A桶*/
27          b1=b1-(a2-a1);a1=a2;
28      }
29      strcat(s,"pore ");strcat(s,B);strcat(s," ");strcat(s,A);strcat(s,"\n");
30  }
31  strcat(s, "success\n");
32  return s;
33 }
```

程序 6-10　实现算法 6-10 的 C 函数

对程序 6-10 的说明如下。

(1) 函数 jugs 有 5 个与算法过程一样的参数。但是,函数并不直接向输出文件写入信息,而是将信息存储于一个字符串中作为返回值。

(2) 与算法过程不同,函数 jugs 不但要处理 x<0 的情形,也要处理 y<0 的情形。两者的区别不仅体现于灌水操作的对象和倒水操作的对象不同,也影响输出信息的不同。所以,要设置变量 a2、b2 来表示倒水、汲水的桶的容量,变量 A、B 表示倒水、汲水的桶标识。x1、y1 表示倒掉水的桶数和灌满水的桶数。并用指针 bb 指向真正的 B 桶中所存有的水量的变量。这些变量根据参数 x 的符号变化,在第 3~12 行进行初始化。

(3) 函数体中第 13~30 行的 **while** 循环实现算法过程中的第 2~14 行的 **while** 循环,模拟自动机在 3 个状态间的转换,直至真正的 B 桶中的水量为 n。在此过程中,把要记录的信息,加载到字符串 s 中。循环结束时,第 31 行在 s 中追加信息“success”。

上述函数 getxy 和 jugs 的定义,以及调用函数 getxy 和 jugs 解决 JUGS 问题的程序存储在文件夹 chap06\Jugs 中的源文件 Jugs.c 中,读者可打开文件研读。

6.4　模运算

若运算限于加法、减法和乘法,模运算与整数的运算一样,不过若运算的模是 n,则每一个计算结果 x 都要被 $\mathbf{Z}_n=\{0, 1, \cdots, n-1\}$ 的元素所替代,即等价于 $x \bmod n$(即用 $x \bmod n$ 替代)。

6.4.1 模加法和乘法

对 $\boldsymbol{Z}_n$ 定义加法和乘法运算并不困难，因为两个整数的等价类唯一地确定其和与积的等价类，即若 $a \equiv a' \pmod n$ 且 $b \equiv b' \pmod n$，则

$$a + b \equiv a' + b' \pmod n$$
$$ab \equiv a'b' \pmod n$$

于是，如下定义模 n 的加法和乘法，记为 $+_n$ 和 $\cdot_n$：

$$[a]_n +_n [b]_n = [a+b]_n$$
$$[a]_n \cdot_n [b]_n = [ab]_n \tag{6-22}$$

也就是说加与乘通常施于代表元素上，不过结果 x 要替换成其类的代表元素（即 $x \bmod n$）。图 6-2(a)给出了 $\boldsymbol{Z}_6$ 上的 $+_6$ 运算表。由此例不难看出：

(1) $\boldsymbol{Z}_n$ 上的加法 $+_n$ 是封闭的，即 $\forall a, b \in \boldsymbol{Z}_n, a +_n b \in \boldsymbol{Z}_n$。

(2) 对于 $\boldsymbol{Z}_n$ 上的加法 $+_n$，元素 0 扮演着"单位元"的角色：任何元素加上 0 仍然为该元素。

(3) 相对于单位元 0，任何元素 x 都有其唯一的"逆元"$-x$。例如，在 $\boldsymbol{Z}_6$ 上，元素 $x=1$ 关于 $+_6$ 的逆元 $-x=5$，而元素 $x=2$ 的加法逆元 $-x=4$。

利用模 n 的乘法定义 $\cdot_n$，我们定义集合 $\boldsymbol{Z}_n^*$：它们由 $\boldsymbol{Z}_n$ 中所有与 n 互质的元素构成，即

$$\boldsymbol{Z}_n^* = \{[a]_n \in \boldsymbol{Z}_n : \mathrm{gdc}(a, n) = 1\}$$

例如 $\boldsymbol{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$，图 6-2(b)给出了 $\boldsymbol{Z}_{15}^*$ 上乘法 $\cdot_{15}$ 的运算表。

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a) $\boldsymbol{Z}_6$上的加法$+_6$运算表

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b) $\boldsymbol{Z}_{15}$上的乘法$\cdot_{15}$运算表

图 6-2 $\boldsymbol{Z}_n$ 上的加法和乘法运算

由此例可见：

(1) $\boldsymbol{Z}_n^*$ 上的乘法 $\cdot_n$ 是封闭的，即 $\forall a, b \in \boldsymbol{Z}_n^*, a \cdot_n b \in \boldsymbol{Z}_n^*$。

(2) 元素 1 是乘法 $\cdot_n$ 的"单位元"，即 $\forall a \in \boldsymbol{Z}_n^*, a \cdot_n 1 = 1 \cdot_n a = a$。

(3) 相对于单位元 1，$\boldsymbol{Z}_n^*$ 上的每个元素 x 都有其唯一的"逆元"x^{-1}。例如，$\boldsymbol{Z}_{15}^*$ 上元素 $x=2$ 关于乘法 $\cdot_{15}$ 的逆元 $x^{-1}=8$，而元素 7 的乘法逆元 $x^{-1}=13$。利用算法 EXTENDED-EUCLID 可以很方便地计算中元素 a 的乘法逆元素：由于 a 与 n 互质，故 EXTENDED-GCD(a, n)返回的(d, x, y)满足：

$$1 = d = ax + ny$$

这蕴含着

$$ax \equiv 1 \pmod n$$

即 $x=a^{-1}$。

作为计算乘法逆元的例子,假定 $a=5,n=11$。那么 EXTENDED-EUCLID(a, n)返回(d, x, y)=(1, −2, 1),所以 $1=5\cdot(-2)+11\cdot 1$。于是,−2(即 9 mod 11)是 5 关于模 11 的乘法逆元。

在本章其余部分中讨论 $\mathbf{Z}_n$ 上加法 $+_n$ 和 $\mathbf{Z}_n^*$ 上乘法 $\cdot_n$ 时,习惯地把等价类表示成它们的代表元素,而将运算符 $+_n$ 和 $\cdot_n$ 分别表示成传统的算术记号+ 和 · 。此外,关于模 n 相等也可以解释为 $\mathbf{Z}_n$ 中的等式。例如,下列两个语句是等价的:

$$ax \equiv b \pmod{n}$$

$$[a]_n \cdot_n [x]_n = [b]_n$$

$\mathbf{Z}_n^*$ 中元素 a 的(乘法)逆表示为(a^{-1} mod n)。$\mathbf{Z}_n^*$ 中的除法用等式 $a/b\equiv ab^{-1} \pmod{n}$ 表示。例如,在 $\mathbf{Z}_{15}^*$ 中,有 $7^{-1}\equiv 13 \pmod{15}$,这是因为 $7\cdot 13\equiv 91\equiv 1 \pmod{15}$,即 $4/7\equiv 4\cdot 13\equiv 7 \pmod{15}$。

集合 $\mathbf{Z}_n^*$ 的元素个数用 $\varphi(n)$ 表示,此函数称为 Eule 的 phi 函数,满足等式

$$\varphi(n) = n\prod_{p|n}\left(1-\frac{1}{p}\right) \tag{6-23}$$

其中,p 取遍所有整除 n 的素数(若 n 是素数,也包含其自身),此处并不证明此等式。直观地,先罗列出 n 的各余数{0, 1, …, $n-1$},然后对每一个整除 n 的素数 p,删除列表中 p 的倍数。例如,由于 45 的素数约数是 3 和 5。

$$\begin{aligned}\varphi(45) &= 45(1-1/3)(1-1/5)\\ &= 45(2/3)(4/5)\\ &= 24\end{aligned}$$

若 p 是素数,则 $\mathbf{Z}_p^*=\{1, 2, \cdots, p-1\}$,且

$$\varphi(p) = p-1 \tag{6-24}$$

若 n 是合数,则 $\varphi(n)<n-1$。

6.4.2 解模线性方程

1. 问题描述与分析

现在来考虑求下列方程解的问题

$$ax \equiv b \pmod{n} \tag{6-25}$$

其中 $a>0,n>0$。此问题是很有用的,例如,在 RSA 公钥密码系统中将其作为寻求钥匙过程的一部分。假定 a、b 和 n 是已知的,要求出所有关于模 n 满足式(6-25)的值 x。可能有零个、一个或多个这样的解。

首先来研究式(6-25)有解的条件,有定理 6-10。

定理 6-10 对未知数为 x 的方程 $ax\equiv b \pmod{n}$ 有解当且仅当 $\gcd(a, n) \mid b$。

证明 设 $\gcd(a, n)=d=x'a+y'n$。先证明充分性,$d|b \Rightarrow \exists k$,使得 $b=kd$。而由此得 $b=kd=k(x'a+y'n)=kx'a+ky'n$。这意味着 kx' 是方程 $ax\equiv b \pmod{n}$ 的一个解。

为说明条件的必要性,设 x_0 是方程 $ax\equiv b \pmod{n}$ 的一个解 $\Rightarrow \exists y_0$,使得 $b=x_0a+y_0n$。由于 $d|a$ 且 $d|n$,根据式(6-9)得 $d|(x_0a+y_0n)$,即 $d|b$。

为在式(6-25)有解的条件下,寻求解的方法,考察下列几个命题。

定理 6-11　设 $d=\gcd(a,\ n)$,并假定有整数 x' 和 y' 使得 $d=ax'+ny'$(例如,通过 EXTENDED-EUCLID 的计算)。若 $d\mid b$,则方程 $ax\equiv b \pmod n$有一个作为它的解的值,满足

$$x_0 = x'(b/d) \bmod n$$

证明　因为有

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod n \\ &\equiv d(b/d) \pmod n \ (\text{因为 } ax' \equiv d \pmod n) \\ &\equiv b \pmod n \end{aligned}$$

于是,x_0 是 $ax\equiv b \pmod n$的一个解。

定理 6-12　假定方程 $ax\equiv b \pmod n$是可解的(即 $d\mid b$,其中 $d=\gcd(a,\ n)$)且 x_0 是该方程的一个解。则此方程关于模 n 恰有 d 个不同的解 $x_i=x_0+i(n/d)$,$i=0,\ 1,\ \cdots,\ d-1$。

证明　由于 $n/d>0$ 且 $0\leqslant i(n/d)<n$,$i=0,\ 1,\ \cdots,\ d-1$,所以值 $x_0,\ x_1,\ \cdots,\ x_{d-1}$关于模 n 均不同余。由于 x_0 是 $ax\equiv b \pmod n$的解,有 $ax_0\equiv b \pmod n$。于是,对 $i=0,\ 1,\ \cdots,\ d-1$,有

$$\begin{aligned} ax_i &= a(x_0+in/d) \\ &= (ax_0+ain/d) \\ &\equiv ax_0 \pmod n \quad (\text{因为 } d\mid a) \\ &\equiv b \pmod n \end{aligned}$$

即 x_i 也是解。

下面说明方程 $ax\equiv b \pmod n$的任一解 x',必与这 d 个解中的一个关于模 n 同余。为此,首先注意到对 $\forall\, i\geqslant d$,根据定理 6-1 存在着 $k,i'\in\mathbf{Z}$,使得 $i=kd+i'$,其中 $0\leqslant i'<d$。而

$$\begin{aligned} x_0+in/d &= x_0+(kd+i')n/d \\ &= x_0+kdn/d+i'n/d \\ &= x_0+kn+i'n/d \\ &\equiv x_0+i'n/d \pmod n \end{aligned}$$

这说明 $\forall\, j\in\mathbf{Z}$,x_0+jn/d 都与 $x_i=x_0+in/d$,$i=0,\ 1,\ d-1$ 之一关于模 n 同余。

由于 $ax'\equiv ax_i \pmod n$,而 $\gcd(a,\ n)=d$,根据推论 6-4,有 $x'\equiv x_i \pmod{n/d}$,$i=0,\ 1,\ d-1$,所以 x'必与 x_0+in/d,$i=0,\ 1,\ d-1$ 之一关于模 n 同余。

2. 算法描述与分析

我们已经在数学上解决了方程 $ax\equiv b \pmod n$,下列的算法将返回该方程所有的解。输入的 a、n 和 b 是任意的正整数。

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
(d, x, y) ← EXTENDED-EUCLID(a, n)
S←∅
if d | b
    then x0 ← x(b/d) mod n
```

```
5          for i ← 0 to d−1
6              do S←S∪{(x0+i(n/d)) mod n }
7   return S
```

算法 6-11　解模线性方程的过程

作为该过程运行的一个例子,考虑方程 $14x \equiv 30 \pmod{100}$(这里,$a=14$,$b=30$,$n=100$)。第 1 行调用 EXTENDED-EUCLID 得到$(d, x, y)=(2, -7, 1)$。由于 $2 \mid 30$,执行第 4~7 行。在第 4 行,计算 $x_0=(-7)(15) \bmod 100=95$。第 5 行和第 6 行的循环得出两个解 95 和 45。

过程 MODULAR-LINEAR-EQUATION-SOLVER 的运行如下。第 1 行计算 $d=\gcd(a, n)$ 以及两个整数值 x 和 y,使得 $d=ax+ny$,说明 x 是方程 $ax \equiv d \pmod{n}$的一个解。若 d 不能整除 b,则根据定理 6-10 方程 $ax\equiv b \pmod{n}$无解。第 3 行检测是否 $d \mid b$;若是,第 4 行按定理 6-11 计算方程 $ax \equiv b \pmod{n}$的一个解 x_0,有了一个解,定理 6-12 断言其余 $d-1$ 个解可由(n/d)关于模 n 的倍数而得。第 5 行和第 6 行的 **for** 循环从 x_0 开始得出所有 d 个解。

MODULAR-LINEAR-EQUATION-SOLVER 执行 $O(\lg n+\gcd(a, n))$次算术运算,这是因为 EXTENDED-EUCLID 执行 $O(\lg n)$次算术运算,第 4 行和第 5 行的 **for** 循环的每次迭代执行常数次算术运算。

定理 6-13 的下列推论给出了 a 与模 n 互质的特殊情形的说明。

推论 6-5　对任一 $n>1$,若 $\gcd(a, n)=1$,则方程 $ax \equiv b \pmod{n}$关于模 n 有唯一的解。

若 $b=1$,这是常用的情形,所求的 x 是关于模 n 的 a 的乘法逆元。

推论 6-6　对任一 $n>1$,若 $\gcd(a, n)=1$,则方程 $ax \equiv 1 \pmod{n}$关于模 n 有唯一的解;否则无解。

3. 程序实现

下面列出实现算法 6-9 的 BigInt 版本。

```
LinkedList* modularEquationSolver(BigInt a, BigInt b, BigInt n){
    LinkedList *S=createList(sizeof(BigInt),NULL);
    BigInt x0, x={NULL,0}, y={NULL,0}, i,
           d=extendedEuclid(a, n,&x, &y),
           t=remainder(b,d),t1,t2;
    if(isZero(t)){
        t1=quotient(b,d);t2=product(x,t1);
        x0=remainder(t2,n);          /* x0←(b/d)x mod n */
        for(clrBigInt(x),i=newIntByint(0);compareInt(&i,&d)<0;increase(&i,1)){
            clrBigInt(&t1),clrBigInt(&t2);clrBigInt(&t);
            t1=quotient(n,d),t2=product(i,t1),t=sum(x0, t2);
            x= remainder(t,n);       /* x←(x0+(n/d)i) mod n */
            listPushBack(S,&x);      /* S←S∪{x} */
        }
```

```
16    }
17    clrBigInt(&y);clrBigInt(&x0);clrBigInt(&d);
18    clrBigInt(&t);clrBigInt(&t1);clrBigInt(&t2);
19    return S;
20 }
```

程序 6-11　实现算法 6-11 的 C 函数 BigInt 版本

对程序 6-11 的说明如下。

(1) 与算法过程一样,函数 modularEquationSolver 有 3 个参数 a、b 和 n,它们都是 BigInt 类型的,表示模方程 ax≡b mod n 中系数 a、常量 b 和模 n。函数返回存储该方程所有解的链表。

(2) 变量 S、x0、x、y、d、i 的意义与算法过程中的同名变量一致。临时变量 t1、t2、t 用来接受调用大整数运算函数返回值,防止内存泄漏。

(3) 第 4 行调用程序 6-8 定义的函数 extendedEuclid,将 d 初始化为 a、n 的最大公约数,并将 x、y 初始化为 d 关于 a、n 的线性组合系数。第 5 行调用函数 remainder 计算 b 除以 d 的余数,并用其来初始化临时变量 t。第 7～16 行的 **if** 语句对应算法过程中第 3～6 行的 **if** 结构。检测的条件是调用函数 isZero 的返回值是否为 0。此函数定义在源文件 bigint.c 中,原型声明为

```
int isZero(BigInt a);
```

该函数判断大整数 a 是否为 0。若是,返回 1,否则返回 0。此处,传递给参数 a 的是表示 b 除以 d 的余数 t,t 为 0 等价于 d|b。若此条件成立,第 8 行和第 9 行完成算法中第 4 行的操作 $x_0 \leftarrow x(b/d) \bmod n$。第 10～15 行的 **for** 循环调用函数 newIntByint(0)初始化 i 为 0,循环条件 compareInt(i,d)<0 相当于 i<d,步进表达式调用函数 increase(&i,1),相当于 i++。循环体中,第 11～14 行完成操作 $x \leftarrow (x_0 + i(n/d)) \bmod n$。第 15 行完成操作 $S \leftarrow S \cup \{x\}$。循环结束时,S 内存储了方程的所有解。

函数 modularEquationSolver 定义于 numbertheory 文件夹中的源文件 lequation.c,原型声明于同一文件内的头文件 lequation.h。读者可打开文件研读实现算法 6-11 函数的整型数据版本。

6.4.3　元素的幂

如同对给定的元素 a 自然地考虑其关于模 n 的倍数一样,自然地会去考虑 a 的关于模 n 的幂构成的序列,其中 $a \in \mathbf{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \cdots \tag{6-26}$$

以 n 为模。从 0 开始,此序列的第 0 个值是 $a^0 \bmod n=1$,且第 i 个值是 $a_i \bmod n$。例如,以 7 为模的 3 的幂是

i	0	1	2	3	4	5	6	7	8	9	10	11	…
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	…

而以 7 为模 2 的幂是

i	0	1	2	3	4	5	6	7	8	9	10	11	…
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	…

1. 重复 B 次方计算幂

数论计算中的一个经常发生的操作是对一个数计算以另一个数为模的幂,这一操作称为**模求幂**。更准确地说,我们希望有一个计算 $a^b \bmod n$ 的有效方法,其中 a、b 是非负整数,n 是正整数。模求幂是很多素数检测程序的基本操作。将 b 表示为二进制形式,可有效地用重复平方法解决此问题。

$$a^b \bmod n$$

设 $b=b_kB^k+b_{k-1}B^{k-1}+\cdots+b_1B+b_0$,则

$$\begin{aligned} a^b &= a^{b_kB^k+b_{k-1}B^{k-1}+\cdots+b_1B+b_0} \\ &= a^{b_kB^k}a^{b_{k-1}B^{k-1}}\cdots a^{b_1B}a^{b_0} \\ &= a^{b_kB^k}a^{b_{k-1}B^{k-1}}\cdots a^{b_1B}a^{b_0} \\ &= (a^{b_k})^{B^k}(a^{b_{k-1}})^{B^{k-1}}\cdots(a^{b_1})^B(a^{b_0}) \end{aligned}$$

```
MODULAR-EXPONENTIATION(a, b, n)
1   d ← 1
2   设(b_k b_{k-1} … b_1 b_0)为 b 的 B 进制表达式
3   for i ← k downto 0
4           do d ← d^B mod n
5            if b_i ≠ 0
6              then d ←(d · a^{b_i})mod n
7   return d
```

算法 6-12 重复 B 次方计算幂的过程

设 b_k, b_{k-1}, …, b_1, b_0 是 b 的二进制表达(即二进制表达的长度为 $k+1$,b_k 是最高位,b_0 是最低位)。下列的过程计算 $a^c \bmod n$,c 以增量 2 从 0 增加到 b。

```
MODULAR-EXPONENTIATION2(a, b, n)
1   d ← 1
2   设< b_k, b_{k-1}, …, b_1, b_0 > 为 b 的二进制表达式
3   for i ← k downto 0
4           do d ← (d · d) mod n
5              if b_i = 1
6                 then d ← (d · a) mod n
7   return d
```

算法 6-13 重复平法计算幂的过程

每次迭代中的第 6 行使用的平方计算是对名称“重复平方”的解释。作为例子,对 $a=7$,$b=561$,以及 $n=561$,算法所计算的以 561 为模的值的序列展示在图 6-3 中;序列的指数

展示在标有 c 的行中。

图 6-3 计算 $a^b(\bmod n)$的结果，其中 $a=7$，$b=560=1000110000$ 以及 $n=561$。d 行展示了每次执行 **for** 循环后的值。最终结果是 1。

i	9	8	7	6	5	4	3	2	1	0
b^i	1	0	0	0	1	1	0	0	0	0
d	7	49	157	526	160	241	298	166	67	1

图 6-3　$a^b(\bmod n)$的结果

变量 c 对算法而言并不是必需的，但将其用于解释之便；算法维护下列包括两部分的循环不变量。

第 4～9 行的 **for** 循环每次重复之前：c 的值与 b 的二进制表达的前缀 b_k，b_{k-1}，…，b_{i+1}相同，且 $d=a^c \bmod n$。

如下使用此循环不变量：

初始：开始时，$i=k$，所以前缀 b_k，b_{k-1}，…，b_{i+1}为空，这对应于 $c=0$。此外，$d=1=a^0 \bmod n$。

维持：设 c'和 d'表示 **for** 的每次重复后的值，当然也是下一次重复前的值。每次重复修改 $c' \leftarrow 2c$（若 $b_i=0$）或 $c' \leftarrow 2c+1$（若 $b_i=1$），所以 c 将在下次重复前是正确的。若 $b_i=0$，则 $d'=d^2 \bmod n=(a^c)^2 \bmod n=a^{2c} \bmod n=a^{c'} \bmod n$，若 $b_i=1$，则 $d'=d^2\, a \bmod n=(a^c)^2 a \bmod n=a^{2c+1} \bmod =a^{c'} \bmod n$，无论如何，下次重复前都有 $d=a^c \bmod n$。

终止：终止时，$i=-1$。于是，$c=b$，这是因为 c 具有 b 的二进制表达的前缀 b_k，b_{k-1}，…，b_0 的值。所以 $d=a^c \bmod n=a^b \bmod n$。

如果输入 a、b 和 n 是二进制 β 位数，则所需的算术运算次数为 $O(\beta)$而所有的位操作次数为 $O(\beta^3)$。

2. 程序实现

B 进制数的模幂运算，与二进制的模幂运算密不可分。这不仅是因为二进制是特殊的 B 进制，更微妙的是可以通过用二进制模幂运算计算算法 6-12 中第 4 行中的 d^B和第 7 行中的 d^{b_i}。所以，在实现算法 6-12 之前，先实现算法 6-13，然后利用前者实现后者。实现代码如下。

```
unsigned long lg2(unsigned long long n){/*计算 n 的二进制位数*/
  unsigned long long x=1;
  unsigned long c=0;
  do{
      c++;
      x=x<<1;
  }while(x<n);
  return c;
}
static BigInt modularPow(BigInt a, long b, BigInt n){
  long i, k=lg2(b),bi=1L<<k;
```

```
12    BigInt d=newIntByint(1),t={NULL,0},t1={NULL,0};
13    if(isOne(a))
14        return d;
15    for(i=k;i>=1;i--){
16        clrBigInt(&t);clrBigInt(&t1);
17        t=product(d,d);t1=remainder(t,n);intAssign(&d,t1);/* d←d² mod n */
18        if(b&bi){                                          /* 若 bᵢ=1 */
19            clrBigInt(&t),clrBigInt(&t1);
20            t=product(d,a);
21            t1=remainder(t, n);
22            intAssign(&d,t1);                              /*  d←d * a mod n */
23        }
24        bi=bi>>1;                                          /* 下一个 bₖ */
25    }
26    clrBigInt(&t);clrBigInt(&t1);
27    return d;
28 }
29 BigInt modularExponent(BigInt a, BigInt b, BigInt n){
30    long i,k=b.value->n,x;
31    BigInt d=newIntByint(1),t={NULL,0},t1={NULL,0};
32    ListNode *bi=b.value->nil->prev;
33    if(isOne(a))
34        return d;
35    for(i=k;i>=1;i--){                                     /* 对 b 的每一位 bᵢ */
36        clrBigInt(&t);
37        t=modularPow(d,Base,n);intAssign(&d,t);            /* d←d^B mod n */
38        x=*((long *)(bi->key));                            /* x=bᵢ */
39        if(x){                                             /* 若 bᵢ≠0 */
40            clrBigInt(&t),clrBigInt(&t1);
41            t=modularPow(a,x,n);t1=product(d, t);
42            clrBigInt(&d);d=remainder(t1,n);               /* d←d * a^x mod n */
43        }
44        bi=bi->prev;
45    }
46    clrBigInt(&t);clrBigInt(&t1);
47    return d;
48 }
```

程序 6-12　实现算法 6-12 重复 B 次方计算幂的 C 函数

程序 6-12 的说明如下。

(1) 第 1～9 行定义的函数 lg2 计算并返回参数 n 的二进制表达式的位数。函数从最低位 c=0,x=2^c起计数,直至 x≥n。返回 c 即为所求。

(2) 第 10～28 行定义的函数 modularPow 实现算法 6-13,重复平方计算由参数 a、b、n 确定的幂 $a^b \bmod n$。

函数中定义的变量 i、k、d、bi 与算法过程中的同名变量意义一致。BigInt 类型的临时变量 t、t1 是进行大整数算术运算时为防止丢弃匿名 BigInt 对象而发生内存泄漏作为中转变量用的。第 11 行将 k 初始化为 b 的二进制长度，bi 初始化为 2^k，用它来析取 b 的二进制表达式中的最高位。第 12 行将 d 初始化为 1。

第 13 行和第 14 行处理底数 a 为 1 的特殊情形。此时，无论 b 是什么数，幂均为 1。第 15～25 行的 **for** 语句，对应算法 6-11 中第 3～6 行的 **for** 循环。其中，第 16 行和第 17 行调用函数 product 和 remainder 完成算法中的 $d \leftarrow (d \cdot d) \bmod n$ 操作。第 7 行检测 bi&b 是否为 1，相当于检测 b 的第 i 位是否为 1。若是，则调用函数 product 和 remainder 完成 $d \leftarrow d \cdot a \bmod n$ 操作。第 24 行执行 bi 右移 1 位的操作 bi＝bi＞＞1，相当于每次重复将 bi 除以 2。

(3) 第 29～48 行定义的函数 modularExponent 实现算法 6-12，重复 B 次方计算由参数 a、b、n 确定的幂 $a^b \bmod n$。由于算法 6-12 与算法 6-13 仅有 b 的进位制表达不同的区别，前者是 B 进制，后者为二进制。所以，仅需把对第 4 行的操作从 $d \leftarrow (d \cdot d) \bmod n$ 的实现换成对操作 $d \leftarrow d^B \bmod n$ 的实现。这只要把计算 $d \cdot d$ 的 product 函数调用换成计算 d^B 的 modularPow 函数调用。类似地，对第 7 行 $d \leftarrow (d \cdot a^{b_i}) \bmod n$ 的操作用函数调用 modularPow 和 product 来实现。另外，需要调用函数 isZero 来实现对 b_i 是否为 0 的检测。代码的说明，与(2)的说明非常相似，读者可对照研读。

为便于代码重用，程序 6-12 中的函数声明于 numbertheory 文件夹中的头文件 exponent.h 中，定义于同一文件夹中的源文件 exponent.c 中。在这两个文件中还包含实现算法 6-10 的整型数据版本，读者可打开文件研读。

6.4.4　应用

青蛙的约会

Description

两只青蛙在网上相识了，它们聊得很开心，于是觉得很有必要见一面。它们很高兴地发现它们住在同一条纬度线上，于是它们约定各自朝西跳，直到碰面为止。可是它们出发之前忘记了一件很重要的事情，既没有问清楚对方的特征，也没有约定见面的具体位置。不过青蛙们都是很乐观的，它们觉得只要一直朝着某个方向跳下去，总能碰到对方的。但是除非这两只青蛙在同一时间跳到同一点上，不然是永远都不可能碰面的。为了帮助这两只乐观的青蛙，要求写一个程序来判断这两只青蛙是否能够碰面，会在什么时候碰面？

把这两只青蛙分别称为青蛙 A 和青蛙 B，并且规定纬度线上东经 0°处为原点，由东往西为正方向，单位长度 1m，这样就得到了一条首尾相接的数轴。设青蛙 A 的出发点坐标是 x，青蛙 B 的出发点坐标是 y。青蛙 A 一次能跳 mm，青蛙 B 一次能跳 nm，两只青蛙跳一次所花费的时间相同。纬度线总长 Lm。现在要求出它们跳了几次以后才会碰面。

Input

输入只包括一行 5 个整数 x、y、m、n、L，其中 $x \neq y < 2000000000$，$0 < m$、$n < 2000000000$，$0 < L < 2100000000$。

Output

输出碰面所需要的跳跃次数,如果永远不可能碰面则输出一行

"Impossible"

Sample Input

1 2 3 4 5

Sample Output

4

1. 问题分析

如图 6-4 所示为周长为 L 的纬线。青蛙 A 从 x 处自东向西每跳 mm,青蛙 B 从 y 处自东向西每跳 nm。两只蛙同时起跳,周而复始,或许 A、B 能在此纬线的某处相遇,或许永远都不会相遇。若能相遇,找出第一次相遇时两蛙掉过的次数 z 并输出,否则输出不能相遇的信息 Impossible。设两蛙跳 z 次后相遇,则有 $x+mz\equiv y+nz \bmod L$。即

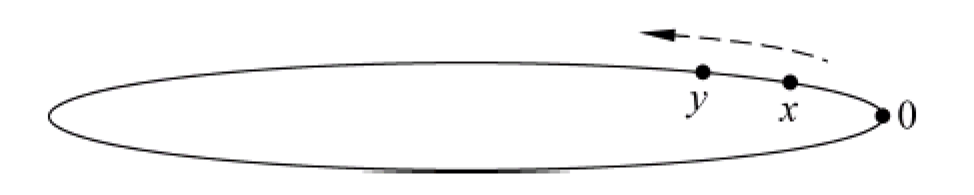


图 6-4　两只青蛙在同一条纬线上自东向西跳跃

$$(m-n)z \equiv y-x \bmod L$$

令 $a=m-n \bmod L$,$b=y-x \bmod L$,则上式为

$$az \equiv b \bmod L$$

根据定理 6-15,$\gcd(a, L)|b$ 是本问题有解的充分必要条件。有解时,最小正数解即为所求。

2. 程序实现

将上述解题思想实现为如下的 C 程序。

```
int main(){
    long long x, y, m, n, L, min, a, b, d;
    LinkedList *s;
    ListNode *p;
    FILE *f1=fopen("chap06/青蛙约会/inputdata.txt","r"),
         *f2=fopen("chap06/青蛙约会/outputdata.txt","w");
    assert(f1&&f2);
    fscanf(f1, "%d %d %d %d %d", &x, &y, &m, &n, &L);
    a= mod(m-n, L); b= mod(y-x, L); d=gcd(a, L);
    if(b%d){                                          /*d  b无解*/
        fprintf(f2, "Impossible\n");
        return 0;
    }
    s=modularLinearEquationSolver(a,b,L);
    min=LONG_MAX;
```

```
16   p=s->nil->next;
17   while(p!=s->nil){                                  /* 寻找最小解 */
18       if(min> *(long long *)(p->key))
19           min= *(long long *)(p->key);
20       p=p->next;
21   }
22   fprintf(f2, "%d\n",min);
23   fclose(f1);fclose(f2);
24   return 0;
25 }
```

程序 6-13　解决青蛙的约会问题的 C 程序

对程序 6-13 的说明如下。

(1) 变量 x、y、m、n、L 分别用来表示青蛙 A 和 B 的起跳位置、一次跳跃的距离及纬线周长。变量 a、b、d 分别表示要解的模线性方程 az≡b mod L 的系数及 a、L 的最大公约数 gcd(a, L)。变量 min 用来跟踪方程的最小解。指针 s 用来指向存放方程解的链表,而指针 p 用来扫描 s 中的结点。

(2) 第 8 行从输入文件中读取 x、y、m、n 和 L 的值。第 9 行计算模线性方程 az≡b mod L 的系数 a 为 m－n mod L, b 为 y－x mod L,而 d 为 gcd(a, L)。第 10～13 行根据定理 6-10 检测该方程是否无解。对有解的情形,第 14 行调用函数 modularLinearEquationSolve 解该线性方程。第 15～21 行扫描存放在 s 指向的链表中的解,跟踪最小者 min。第 23 行将解写入输出文件。

(3) 第 14 行调用的函数 modularLinearEquationSolve 是实现算法 6-11 的整型数据版本,该函数定义于文件夹 numbertheory 中的源文件 lequation. c 中,原型声明于同一文件夹的头文件 lequation. h 中,读者可打开文件研读。

6.5　素数检测

本节中,来考虑寻求大素数的问题。从讨论合数的判断方法开始,进而研究素数的密度,接着考察一个可靠但不完整的检测方法,然后介绍一个由 Miller 和 Rabin 提出的有效的随机素数测试法。

6.5.1　伪素数检测

引理 6-4　n 为正整数,$\mathbf{Z}_n^*=\{z_1, z_2, \cdots, z_{\varphi(n)}\}$。$\forall a\in \mathbf{Z}_n^*$,序列

$$z_1 a,\ z_2 a,\ \cdots,\ z_{\varphi(n)} a$$

关于模 n 是序列

$$z_1, z_2, \cdots, z_{\varphi(n)}$$

的一个排列。

证明 由于 $\gcd(a, n)=1$,故不存在整数 $i(i=1,2,\cdots,\varphi(n),)$使得 $z_ia \equiv 0 \pmod n$。这意味着所有 $z_ia(i=1,2,\cdots,\varphi(n),)$均与序列 $z_1,z_2,\cdots,z_{\varphi(n)}$ 中之一元素关于模 n 同余。下证诸 $z_ia, i=1,2,\cdots,\varphi(n)$关于模 n 两两不同余。若不然,有 $1\leqslant i,j\leqslant\varphi(n)$,且 $i\neq j$ 而使得 $z_ia\equiv z_ja \pmod n$。由于 $\gcd(a, n)=1$,根据推论 6-4 知 $z_i\equiv z_j \pmod n$,这是一个矛盾。结论因此得证。

定理 6-13(Euler 定理) 对任一整数 $n>1$,对所有的 $a\in \boldsymbol{Z}_n^*$

$$a^{\varphi(n)} \equiv 1 \pmod n$$

其中,$\varphi(n)$是 6.4.1 节定义的 Eule 的 phi 函数。

证明 根据引理 6-4,序列 $z_1a, z_2a, \cdots, z_{\varphi(n)}a$ 关于模 n 是序列 $z_1,z_2,\cdots,z_{\varphi(n)}$ 的一个排列。所以,根据式(6-6)有 $z_1a\cdot z_2a\cdot\cdots z_{\varphi(n)}a\equiv z_1\cdot z_2\cdots z_{\varphi(n)} \pmod n$,即 $a^{\varphi(n)}(z_1\cdot z_2\cdots z_{\varphi(n)})\equiv(z_1\cdot z_2\cdots z_{\varphi(n)}) \pmod n$。由于 $\gcd((z_1\cdot z_2\cdots z_{\varphi(n)}), n)=1$,根据推论 6-4 有 $a^{\varphi(n)}\equiv 1 \pmod n$。

推论 6-7(Fermat 定理) 若 p 是素数,则对任意的 $a\in \boldsymbol{Z}_p^*$

$$a^{p-1} \equiv 1 \pmod p$$

此推论适用于 $\boldsymbol{Z}_p$ 中除了 0 以外的每个元素,这是因为 $0\notin \boldsymbol{Z}_p^*$。因此,若 p 为素数,对所有的 $a\in \boldsymbol{Z}_p(a\neq 0)$, $a^p\equiv a \pmod p$。

利用 Fermat 定理,来考虑一个"基本能工作"的素数检测方法,事实上这个方法在很多实际应用中是够好的了。对消除此方法中小毛病的更精练的算法将在稍后介绍。设 $\boldsymbol{Z}_n^+$ 表示 $\boldsymbol{Z}_n$ 中的非零元素:

$$\boldsymbol{Z}_n^+ = \{1,2,\cdots,n-1\}$$

若 n 是素数,则 $\boldsymbol{Z}_n^+ = \boldsymbol{Z}_n^*$。

称合数 n 是**基**-$\boldsymbol{a}$ **的伪素数**,若

$$a^{n-1} \equiv 1 \pmod n \tag{6-27}$$

Fermat 定理(推论 6-7)蕴含着若 n 是素数,则 $\boldsymbol{Z}_n^+$ 中的每一个元素 a、n 都满足式(6-27)。于是,若能找到一个 a 使得 n 不满足式(6-27),则 n 必是合数。令人惊讶的是,反之几乎成立。所以此判断标准几乎是完美的素数测试。检测 n 对 $a=2$ 是否满足式(6-27):如果不满足,断定它是合数,否则,输出 n 是素数的猜想(事实上,此时知道 n 要么是素数,要么是一个基-2 的伪素数)。

下列的过程按此种方式检测 n 是否为素数。它利用 6.4.4 节的过程 MODULAR-EXPONENTIATION。输入的 n 假定是一个大于 2 的奇数。

```
PSEUDOPRIME(n)
1 if MODULAR-EXPONENTIATION(2, n-1, n)≡1 (mod n)
2        then return PRIME           ▷我们希望
3        else return COMPOSITE       ▷明确地
```

算法 6-14 利用 Fermat 定理的伪素数判定过程

这个过程只在一种情况下可能会出错。也就是说,若它说 n 是合数,则它总是对的。但若它说 n 是一个素数,而 n 是一个基-2 的伪素数时就产生一个错误。

该过程出错的机会有多少呢?出奇的少。小于 10000 的 n 中只有 22 个值会发生错误;

最先的 4 个值是 341、561、645 和 1105。可以证明，当 $\beta \to \infty$ 时，该程序运行于随机 β 二进制位整数出错的概率趋于零。出于实用的考虑，如果在一些应用中仅仅是寻求一个大素数，则随机地选择大整数并调用 PSEUDOPRIME 直至对所选整数输出"素数"为止几乎绝不出错。若所测试的整数不是随机选取的，需要有更好的素数检测方法。

不幸的是，不能对接下来的基数（例如，$a=3$）完全估计出直接检测式（6-27）的错误，因为存在对所有的 $a \in \mathbf{Z}_n^+$ 满足式（6-27）的合数 n，这样的整数称为 Carmichael 整数。开头的 3 个 Carmichael 是 561、1105 和 1729。Carmichael 数非常稀少；例如，只有 255 个小于 100000000 的 Carmichael 数。下面说明如何改进素数检测方法使得不会被 Carmichael 数愚弄。引入以下概念：

设 $g \in \mathbf{Z}_n^*$，若序列

$$g, g^2, \cdots, g^{\varphi(n)}$$

关于模 n 是序列

$$z_1, z_2, \cdots, z_{\varphi(n)}$$

的一个排列，则称 g 是 $\mathbf{Z}_n^*$ 的一个**原根**。

例如，3 是以 7 为模的一个原根，但 2 不是。不是对任意的正整数 n，$\mathbf{Z}_n^*$ 都有原根。我们不加证明地给出定理 6-14，它指出了使得 $\mathbf{Z}_n^*$ 存在原根的正整数 n 的特征。

定理 6-14　使 $\mathbf{Z}_n^*$ 具有原根的整数值 $n(n>1)$ 是 $2,4,p^e$ 和 $2p^e$，其中 p 是大于 2 的任意素数，e 是任意的正整数。

若 g 是 $\mathbf{Z}_n^*$ 的一个原根且 a 是 $\mathbf{Z}_n^*$ 的任一元素，则存在一个 z 使得 $g^z \equiv a \pmod n$。此 z 称为以 n 为模 a 的对基底 g 的**离散对数**或 a 的**指数**；将此值表示为 $\mathrm{ind}_{n,g}(a)$。

定理 6-15（离散对数定理）　若 g 是 $\mathbf{Z}_n^*$ 的一个原根，则等式 $g^x \equiv g^y \pmod n$ 成立当且仅当等式 $x \equiv y \pmod{\varphi(n)}$ 成立。

证明　首先假定 $x \equiv y \pmod{\varphi(n)}$，则有某整数 k，使得 $x=y+k\varphi(n)$。所以，

$$\begin{aligned} g^x &\equiv g^{y+k\varphi(n)} \pmod n \\ &\equiv g^y \cdot (g^{\varphi(n)})^k \pmod n \\ &\equiv g^y \cdot 1^k \pmod n \ (\text{根据 Euler' 定理}) \\ &\equiv g^y \pmod n \end{aligned}$$

反之，假定 $g^x \equiv g^y \pmod n$。因为 g 的幂序列是 $\mathbf{Z}_n^*$ 中所有元素构成序列的一个排列，所以，若 $g^x \equiv g^y \pmod n$，则必有 $x \equiv y \pmod{\varphi(n)}$。

离散对数能简化模等式的便利之处示例于定理 6-16 的证明之中。

定理 6-16　若 p 是一个奇素数且 $e \geqslant 1$，则方程

$$x^2 \equiv 1 \pmod{p^e} \tag{6-28}$$

有且仅有两个解，即 $x=1$ 和 $x=-1$。

证明　设 $n=p^e$。定理 6-14 蕴含着 $\mathbf{Z}_n^*$ 有一个原根 g。式（6-28）可以写成

$$(g^{ind_{n,g}(x)})^2 \equiv g^{ind_{n,g}(1)} \pmod n \tag{6-29}$$

注意 $\mathrm{ind}_{n,g}(1)=0$，定理 6-15 意味着式（6-29）等价于

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\varphi(n)} \tag{6-30}$$

为解此方程中的未知数 $\mathrm{ind}_{n,g}(x)$，运用 6.4 节中的方法。根据式（6-23）有 $\varphi(n)=p^e(1-1/p)=(p-1)\ p^{e-1}$，设 $d=\gcd(2,\varphi(n))=\gcd(2,(p-1)\ p^{e-1})=2$，并注意 $d \mid 0$，

由定理 6-12 知道式(6-30)恰有 $d=2$ 个解。所以,式(6-28)恰有 2 个解,观察可得 $x=1$ 和 $x=-1$。

数 x 是以 n 为模 1 的一个**非平凡平方根**,若它满足方程 $x^2\equiv 1 \pmod n$,但 x 对模 n 而言,既不等价于 1 也不等价于 −1。例如,6 是以 35 为模的 1 的非平凡平方根。推论 6-8 将用于 6.5.2 节中的 Miller-Rabin 的素数测试过程的正确性证明。

推论 6-8 若存在以 n 为模 1 的非平凡平方根,则 n 是合数。

证明 根据定理 6-16 的逆否命题,若存在以 n 为模 1 的非平凡平方根,则 n 不可能是一个奇素数或奇素数的幂。若 $x^2\equiv 1 \pmod 2$,则 $x\equiv 1 \pmod 2$,所以,以 2 为模 1 的所有平方根都是平凡的。于是,n 不可能是素数。最后,对 1 的非平凡平方根,必有 $n>1$。所以,n 必为合数。

6.5.2 Miller-Rabin 的随机素数检测

Miller-Rabin 的素数检测在以下两个方面做了修改后克服了 PSEUDOPRIME 简单检测的问题。

(1) 它要进行若干次随机选定的基值的测试,而不仅仅是对一个基值进行。

(2) 在计算每一个模求幂的同时,它关注是否在最后的平方集合中发现以 n 为模的 1 的一个非平凡平方根。若是,则停止并返回 COMPOSITE。推论 6-8 证明在此方式中检测到的是合数。

Miller-Rabin 的伪码做如下素数检测。输入($n>2$)是一个被测试的奇数,s 是要随机选取的用于测试的基值个数。代码利用随机数发生器 RANDOM(1, $n-1$)返回一个满足 $1\leqslant a\leqslant n-1$ 的整数 a。代码还利用一个辅助过程 WITNESS,使得 WITNESS(a, n)为 TRUE 当且仅当 a 是合数 n 的一个“证词”,即若 a 可能作为 n 是合数的一个证明(以我们将看到的方式)。WITNESS(a, n)是测试

$$a^{n-1}\not\equiv 1 \pmod n$$

的一个扩展但比它更有效,而该测试形成 PSEUDOPRIME 的一个基(用 $a=2$)。先介绍说明 WITNESS 的构建,然后说明它是如何用于 Miller-Rabin 的素数检测的。设 $n-1=2^t u$,其中 $t\geqslant 1$ 且 u 是一个奇数,即 $n-1$ 的二进制表示是奇数 u 的二进制表示后跟 t 个零。所以,$a^{n-1}\equiv (a^u)^{2^t} \pmod n$,于是可以通过计算 $a^u \bmod n$ 并连续地求 t 次平方来计算 $a^{n-1} \bmod n$。

```
WITNESS(a, n)
设 n−1=2^t u, 其中 t ≥1 且 u 是奇数
x_0 ← MODULAR-EXPONENTIATION(a, u, n)
for i←1 to t
    do x_i ← x_{i−1}^2 mod n
       if x_i =1 且 x_{i−1}≠1 且 x_{i−1}≠n−1
           then return TRUE
if x_t≠1
    then return TRUE
return FALSE
```

算法 6-15 提供合数证据的过程

此 WITNESS 伪代码通过在第 2 行计算值 $x_0 = a^u \bmod n$，然后通过第 3～6 行的 **for** 循环将此结果做 t 次平方来计算 $a^{n-1} \bmod n$。通过对 i 的递推，计算所得的值序列 $x_0, x_1, \cdots, x_t$ 满足等式 $x_i \equiv a^{2^i u} \pmod{n}$，$i=0, 1, \cdots, t$。特殊地，有 $x_t \equiv a^{n-1} \pmod{n}$。只要执行了第 4 行的平方运算，若在第 5 行和第 6 行中检测到 1 的一个非平凡平方根，循环就会提前终止。如果是这样，该算法停止并返回 TRUE。在第 7 行和第 8 行中若计算得的 $x_t \equiv a^{n-1} \pmod{n}$ 不等于 1，如过程 PSEUDOPRIME 返回 COMPOSITE 那样返回 TRUE。如果没有在第 6 行或第 8 行返回 TRUE，则在第 9 行返回 FALSE。

现在认为，若 WITNESS(a, n)返回 TRUE，则用 a 构造了一个 n 为合数的证据。

如果 WITNESS 由第 8 行返回 TRUE，则它发现 $a^{n-1} \bmod n \neq 1$。若 n 是素数，根据 Fermat 定理（推论 6-7），对所有的 $a \in \mathbf{Z}_n^+$，$a^{n-1} \equiv 1 \pmod{n}$。所以 n 不可能是素数，且式子 $a^{n-1} \bmod n \neq 1$ 是此事实的一个证明。

若 WITNESS 由第 6 行返回 TRUE，则它发现 x_{i-1} 是以 n 为模，$x_i = 1$ 的一个非平凡平方根，这是因为 $x_{i-1} \equiv \pm 1 \pmod{n}$ 且 $x_i \equiv {x_{i-1}}^2 \equiv 1 \pmod{n}$。推论 6-8 断言仅当 n 是一个合数时，才可能有以 n 为模的 1 的非平凡平方根。所以，是以 n 为模的 1 的一个非平凡平方根的实例是 n 为合数的一个证明。

这样就完成了 WITNESS 的正确性的证明。若调用 WITNESS(a, n)输出 TRUE，则 n 当然是合数，且 n 是合数的证明可很容易地由 a 和 n 确定。

现在来考察基于 WITNESS 的 Miller-Rabin 的素数检测。在此假设 n 是一个大于 2 的奇数。

```
MILLER-RABIN(n, s)
1  for j ← 1 to s
2      do a ← RANDOM(1, n−1)
3         if WITNESS(a, n)
4            then return COMPOSITE      ▷按定义
5  return PRIME                         ▷几乎肯定
```

算法 6-16　检测素数的随机算法过程

过程 MILLER-RABIN 是对 n 是合数的证词的随机搜索。主循环（从第 1 行开始）从 $\mathbf{Z}_n^+$ 中选取随机值（第 2 行）。若选取的 a 中有一个是合数 n 的证词，则 MILLER-RABIN 在第 4 行输出 COMPOSITE。根据 WITNESS 的正确性，这样的输出总是对的。若在 s 次尝试中没有发现证词，MILLER-RABIN 因此假定没有证词被发现，并假定 n 是素数。将看到，当 s 足够大时，此输出很可能是正确的。然而，在很小的机会中，该过程会不幸地在其所选的 a 中没发现证词但却存在有证词。

为示例 MILLER-RABIN 的操作，设 n 是 Carmichael 数 561，所以 $n-1=560=24 \cdot 5$。假定 $a=7$ 并将其选为基，图 6-3 展示了 WITNESS 计算 $x_0 \equiv a^{35} \equiv 241 \pmod{561}$，并由此计算出序列 $X = \langle 241, 298, 166, 67, 1 \rangle$。于是，由于 $a^{280} \equiv 67 \pmod{n}$ 及 $a^{560} \equiv 1 \pmod{n}$，在最后一次求平方步骤中发现一个 1 的非平凡平方根。所以，是 n 为合数的一个证词，WITNESS$(7, n)$ 返回 TRUE，且 MILLER-RABIN 返回 COMPOSITE。

若 n 是一个 β 二进制位整数，MILLER-RABIN 需要 $O(s\beta)$ 次算术运算和 $O(s\beta^3)$ 次位运

算,这是因为它渐进地需要不超过 s 次模求幂运算。

6.5.3 Miller-Rabin 素数检测的错误率

若 MILLER-RABIN 输出 PRIME,则有一个很小的出错机会。不像 PSEUDOPRIME,出错的几率独立于 n;对此过程而言,没有坏的输入。相反,它依赖于 s 的大小和选取基值 a 时的“运气”。由于每次对式(6-27)的检测更严格,可以按通常的理由期望对随机选择的整数 n 错误率应当是很小的。定理 6-17 提供了更准确的论据。

定理 6-17 若 n 是一个奇合数,则 n 是合数的证词个数至少为 $(n-1)/2$。

定理 6-18 对任意的奇整数 $n>2$ 和正整数 s,MILLER-RABIN(n, s)出错的概率至多为 2^{-s}。

证明 利用定理 6-17,我们知道若 n 是一个合数,则算法 6-16 第 1～4 行得 **for** 循环的每次执行至少以 1/2 的概率发现一个 n 为合数的证词。MILLER-RABIN 只可能在其主循环的所有的 s 次重复中都不幸地没有发现 n 是合数证词的情况下才出错,这样的一串错误的概率为 2^{-s}。

于是,选择 $s=50$,就几乎能满足任意可想象的应用。

6.5.4 程序实现

1. 随机数发生器

算法 6-16 的 MILLER-RABIN 过程运行时,需要调用随机数发生器 RANDOM 产生一个落于区间[1, $n-1$)内的随机整数。C 语言提供了一个库函数 rand(),它声明于头文件 stdlib.h 中。调用该函数产生并返回一个落于区间[0, RAND_MAX]内的随机整数。因此,需要改造该函数,使得能对任意大整数 n,产生一个介于 0～n－1 之间的整数。为此,先定义如下函数。

```
typedef unsigned long long ul_int;
typedef long long l_int;
ul_int RangedRandom( ul_int min, ul_int max){
    ul_int u=(double)rand()/(RAND_MAX+1) * (max-min)+min;
    return u;
}
```

程序 6-14 产生介于给定的两个整型数据 min 和 max 之间的随机整数的 C 函数

对由 **unsigned long long** 型参数 min、max(前者小于后者),RangedRandom 函数调用 rand()产生一个[0, RAND_MAX]之间的整数,转换成 **double** 类型后除以 RAND_MAX+1 后得到一个介于[0, 1)的 **double** 型随机数。用此数乘以(max－min)并加上 min 就得到一个介于[min, max)的随机数,将其整数部分作为函数值返回。利用该函数,对任意指定的大整数 a 可定义下列的产生介于[0, a)之间的随机整数的函数。

```
1 BigInt random(BigInt a){
2   BigInt b={createList(sizeof(long),NULL),0};
3   ListNode *p=a.value->nil->next;            /* a 的最低位 */
4   long x;
5   assert(a.sign==0);
6   srand( time(NULL) );
7   x=RangedRandom(1, *((long *)(p->key)));/* 产生 b 的最低位 */
8   listPushBack(b.value,&x);
9   p=p->next;
10   while(p!=a.value->nil){                   /* 自低向高产生 b 的每一位 */
11       x=RangedRandom(0, *((long *)(p->key)));
12       listPushBack(b.value,&x);
13       p=p->next;
14   }
15   p=b.value->nil->prev;                    /* 从 b 的最高位起去除高位 0 */
16   while(p!=b.value->nil->next&&( *((long *)(p->key))==0)){
17       listDelete(b.value,p);
18       p=b.value->nil->prev;
19   }
20   return b;
21 }
```

程序 6-15　产生随机大整数的 C 函数

对程序 6-15 的说明如下。

(1) 函数 random 产生一个介于 0～a 之间的随机 BigInt 型大整数,并返回。

(2) 函数中设置 BigInt 型变量表示要创建随机数,ListNode 型指针 p 用来访问表示 a 的绝对值每一位数的链表结点。**long** 型变量 x 用来存放所产生的随机数的每一位数。

(3) 第 7 行调用函数 RangedRandom(1, *((**long** *)(p->key)))产生介于 0～p 指向的 a 的最低位之间的随机数 x,第 8 行调用 listPushBack(b.value,&x)将 x 作为 b 的最低位。第 10～14 行的 **while** 循环逐位产生介于 0～a 的当前位的随机数,作为 b 的当前位。由于所产生的随机数可能为 0,第 15～19 行负责去除 b 的高位 0。

2. 素数检测

算法 6-16 的 MILLER-RABIN 过程运行时需要调用算法 6-15 的 WITNESS 过程为 a 是合数提供证词。下列函数实现 WITNESS 过程。

```
int Witness(BigInt a, BigInt n){
  BigInt u={NULL,0},x0,xi={NULL,0},tow=newIntByint(2),n1={NULL,0};
  long r,t=0,i;
  intAssign(&u,n),decrease(&u,1);intAssign(&n1,u);/* u、n1 初始化为 n-1 */
  do{
      t++;
      dividedByDigit(u.value,2,&r);
  } while(! isZero(u)&&! r)
```

```
9  x0=modularExponent(a, u, n);
10  for(i=1;i<=t;i++){
11      clrBigInt(&xi);
12      xi=modularExponent(x0,tow,n);
13      if(isOne(xi)&&! isOne(x0)&&compareInt(&x0,&n1))
14          return 1;
15      intAssign(&x0,xi);
16  }
17  if(!isOne(xi))
18      return 1;
19  return 0;
20 }
```

程序 6-16　实现算法 6-15 的 C 函数 BigInt 版本

对程序 6-16 的说明如下。

(1) 函数 Witness 实现算法 6-15 的 WITNESS 过程,对参数 n 表示的整数判断由参数 a 提供的 n 为合数证词是否正确。若 a 足以证明 n 是合数,函数返回 1,否则返回 0。

(2) 函数中设置的变量 u、t、x0、xi 与算法过程中的同名变量意义一致。n1 用来表示 n−1。

(3) 第 5～8 行通过一个 **do-while** 循环实现算法中的第 1 行,将 n−1 分解为 2^tu。在循环体中,累加 2^t的指数 t,调用函数 dividedByDigit(u. value,2,&i)计算 u(第 4 行初始化为 n−1)除以 2 的商。该函数定义于源文件 bigint. c 中。注意,其中的第 1 个参数是 u 的成员 value,商就地存储于 value 中。第 3 个 r 参数是用来保存余数的。循环的条件是 u>1 且 r=0。

第 9 行调用程序 6-16 定义的函数 modularExponent(a, u, n)执行操作 x_0 ← MODULAR-EXPONENTIATION(a, u, n),第 10～16 行的 **for** 语句对应算法过程中第 3～6 行的 **for** 循环,迭代计算 xi。第 13 行调用函数 isOne(xi)、isOne(x0)和 compareInt(&x0, &n1)检测条件 $x_i=1$ 且 $x_{i-1}\neq 1$ 且 $x_{i-1}\neq n-1$。函数 isOne 和 compareInt 定义于源文件 bigint. c 中,分别用来判断大整数的值是否为 1,即比较两个大整数的大小关系。

```
int MillerRabin(BigInt n, int s){
    BigInt a={NULL,0};
    int j;
    for(j=0;j<s;j++){
        clrBigInt(&a);
        a=random(n);
        if(Witness(a,n)){
            clrBigInt(&a);
            return 0;
        }
    }
    clrBigInt(&a);
```

```
13    return 1;
14 }
```

程序 6-17　实现算法 6-16 MILLER-RABIN 过程 C 函数的 BigInt 版本

函数 MillerRabin 实现算法 6-16 的 MILLER-RABIN 过程，以 1－1/2s 的概率判断大整数 n 是否为素数。若 n 是素数，函数返回 1，否则返回 0。

函数中定义的变量 a、j 与算法过程中的同名变量意义一致。

第 4～11 行的 **for** 语句对应算法过程中第 1～4 行的 **for** 循环。其中，第 5 行和第 6 行调用程序 6-15 定义的函数 random(n)实现 $a \leftarrow$ RANDOM(1，$n-1$)操作。第 7～10 行调用程序 6-16 定义的函数 Witness(a，n)检测 a 是否为 n 是合数的证词，若是，则返回 0。循环重复 s 次完成，则判断 n 为素数，返回 1。

程序 6-14～程序 6-17 罗列的函数均定义于 numbertheory 文件夹的源文件 primetest. c 中，函数 MillerRabin 的原型声明于同一文件夹的头文件 primetest. h 中。在这两个文件中还包含了实现 MILLER-RABIN 过程的 C 函数的整型数据版本，读者可打开文件研读。

6.6　整数分解

假定有一个要做素因数分解的整数 n，即要将 n 分解成素数之积。6.5.4 节的素数检测将告诉人们 n 是合数，但它不能告诉人们 n 的素数因子。分解大整数 n 要比确定 n 是否为素数困难得多。迄今为止，用当今的超级计算机和最好的算法对任意 1024 位二进制位的整数进行因数分解都是不可行的。最直接的计算整数因数分解的算法是试商法。设 n 是一个合数。

```
TRY-DIVISION(n)
for d←2 to √n
    do if d|n
        then return d
```

算法 6-17　用试商法计算整数因子的过程

过程 TRY-DIVISION 可以保证在$\sqrt{n}$次试商后得到n 的一个因数。设 n 的位数为β，则运行时间为 $\Theta(\beta^2\sqrt{n})=\Theta(2\beta^2 2^{\beta/2})=O(2^{\beta/2})$。

6.6.1　Pollard 的ρ 探索法

本节讨论一个用与 TRY-DIVISION 同样的工作量，计算 n^2 内的整数因数(除非运气不佳)的随机过程。

```
POLLARD-RHO(n)
i ← 1
x1 ← RANDOM(0, n-1)
```

```
3   S ← ∅
4   y ← x1
5   k ← 2
6   repeat
7      S ← S ∪ { xi }
8      d ← gcd(|y − xi|, n)
9      if d ≠ 1 and d ≠ n
10         then return d
11     i ← i+1
12     if i=k
13         then y ← xi
14              k ← 2k
15     xi ← (x²i−1 − 1) mod n
16  until xi ∈ S
17 return n
```

算法 6-18　计算合数因数的随机过程

此过程的运行如下。第 1～3 行初始化 i 为 1,随机地在 $\mathbf{Z}_n$ 中选取一值给 x_1,并将集合 S 置为空。从第 6 行开始的 **repeat** 循环一直重复直至新得到的 x_i 的值在集合 S 中重复,搜索 n 的因数。在此 **repeat** 循环的每次重复时,利用第 15 行的迭代

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{6-31}$$

产生无穷序列

$$x_1, x_2, x_3, x_4, \cdots \tag{6-32}$$

中的下一个 x_i 的值,对应的 i 的值在第 11 行得到增加。为了清晰,代码中使用带有下标的变量 x_i,但取消下标程序也一样工作。这是因为所要维护的仅仅是当前的 x_i 值。做此修改该过程只需常数大小的内存。

程序偶尔将当前 x_i 的值保存在变量 y 中。具体地说,被保存的是下标为 2 的幂的值:

$$x_1, x_2, x_4, x_8, x_{16}, \cdots$$

第 4 行保存 x_1 的值,第 13 行当 i 等于 k 时,保存 x_k。变量 k 在第 5 行中初始化为 2,每当 y 被修改时,第 14 行将 k 加倍。所以,k 按序列 1, 2, 4, 8 取值,且总是给出下一个要保存到 y 中的 x_k 值得下标。

第 8～10 行利用保存在 y 中的值和当前的 x_i 值,试图寻求 n 的一个因数。具体地说就是在第 8 行计算的最大公约数 $d=\gcd(|y-x_i|, n)$。若 d 是 n 的一个非平凡约数(第 9 行测试),则第 10 行返回 d。

POLLARD-RHO 可能输出 n 本身,不能保证它必得到 n 的非平凡因数。需要注意的是 POLLARD-RHO 绝不会输出错误的答案,它所返回的数 d 必是 n 的一个非平凡约数。我们将看到,有理由期望 POLLARD-RHO 在经过其中的 **while** 循环的 $\Theta(\sqrt{n})$ 次重复就打印出 n 的一个素因数 p。于是,若 n 是一个合数,由于 n 的每一个素因数 p 都可以期望它小于 $\sqrt{n}$,可以期望此过程在进行了 $n^{1/4}$ 次重复后能够发现完成 n 的因数分解的足够多的约数。

由于 $\mathbf{Z}_n$ 是有限集合,且序列(6-32)中的每一个值仅依赖于其前一个值,所以序列(6-32)终究会重复。一旦得到一个 x_i 使得有某个 $j<i$,有 $x_i=x_j$,就进入一个周期循环。

这是因为 $x_{i+1}=x_{j+1}$，$x_{i+2}=x_{j+2}$ 等。名称 ρ 探索法是缘于如图 6-4 所展示的序列 x_1，x_2，…，x_{j-1} 形成 ρ 的尾巴，而周期循环 x_j，x_{j+1}，…，就形成 ρ 的圈。

先来考虑对 x_i 而言发生重复的序列长度的问题。为估计这个长度，假定函数

$$f_n(x)=(x^2-1) \bmod n$$

的行为如同一个随机函数，虽然这不是真正随机的。这样可以认为每个 x_i 是按均匀分布独立地从 $\mathbf{Z}_n$ 中选取的。根据概率论的"生日悖论"的分析，在得到周期循环前期望的步骤是 $\Theta(\sqrt{n})$。

接下来，考虑做一些修改。设 p 是 n 的非平凡因数使得 $\gcd(p, n/p)=1$。例如，若 n 有因数分解 $n=p_2^{e_2}p_3^{e_3}\cdots p_r^{e_r}$，则可以取 p 为 $p_1^{e_1}$。

序列 $\langle x_i\rangle$ 将蕴含对应的以 p 为模的序列 $\langle x_i'\rangle$，其中

$$x_i'=x_i \bmod p, i=1,2,\cdots$$

因为只用以 n 为模的算术运算（平方和减法）来定义 f_n，我们将看到，可以由 x_i' 算得 x_{i+1}'；该序列"以 p 为模"的观点是以 n 为模的较小的版本。

$$\begin{aligned}
x_{i+1}' &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2-1) \bmod n) \bmod p \\
&= (x_i^2-1) \bmod p \\
&= ((x_i \bmod p)^2-1) \bmod p \\
&= ((x_i')^2-1) \bmod p \\
&= f_p(x_i')
\end{aligned}$$

于是，尽管没有显式地计算序列 $\langle x_i'\rangle$，该序列是根据与序列 $\langle x_i\rangle$ 一样的迭代良好定义的。

图 6-5(a)为从 $x_1=2$ 开始的，按迭代 $x_{i+1}\leftarrow(x_i^2-1) \bmod 1387$ 产生的各个值。1387 的素因数分解是 19 · 73。粗箭头表示因数 19 发现前的迭代次数。细箭头指出在迭代中未达到的各个值，并表示成 ρ 的形状。带有阴影的是被 POLLARD-RHO 存储到 y 的值。因数 19 在得到 $x_7=177$ 时，由 $\gcd(63-177, 1387)=19$ 计算而得。第一个重复的 x 值是 1186，在此值重复之前，因数 19 已经发现。图 6-5(b)是以 19 为模的同样的迭代产生的值。在图 6-5(a)中给出的每一个值 x_i，是与此处的以 19 为模 x_i' 等价的。例如，$x_4=63$ 和 $x_7=177$ 都等价于以 19 为模的 6 的（意为 $63\equiv6$ MOD 19，$177\equiv6$ MOD 19）。图 6-5(c) 是以 73 为模的同样的迭代产生的值。在图 6-5(a)中给出的每一个值 x_i，是与此处的以 19 为模 x_i'' 等价的。根据中国剩余定理，图 6-5(a)中的每一个结点对应一对分别来自于图 6-5(b)和图 6-5(c)的结点。

与前面一样的理由，我们发现序列 $\langle x_i'\rangle$ 重复前的步骤数为 $\Theta(\sqrt{n})$。若 p 相对于 n 较小，序列 $\langle x_i'\rangle$ 发生重复可能比 $\langle x_i\rangle$ 快得多。事实上，只要序列 $\langle x_i\rangle$ 中的两个元素关于模 p 相等，而不是关于模 n，序列 $\langle x_i'\rangle$ 就发生重复。作为示例，如图 6-5(b)和图 6-5(c)所示。

设 t 表示序列 $\langle x_i'\rangle$ 中第一个重复值的下标，并设 $u>0$ 表示所产生的周期循环的长度，即 t 和 $u>0$ 是使得对所有的 $i\geqslant0$，$x_{t+i}'=x_{t+u+i}'$ 的最小的值。根据以上讨论，t 和 u 的期

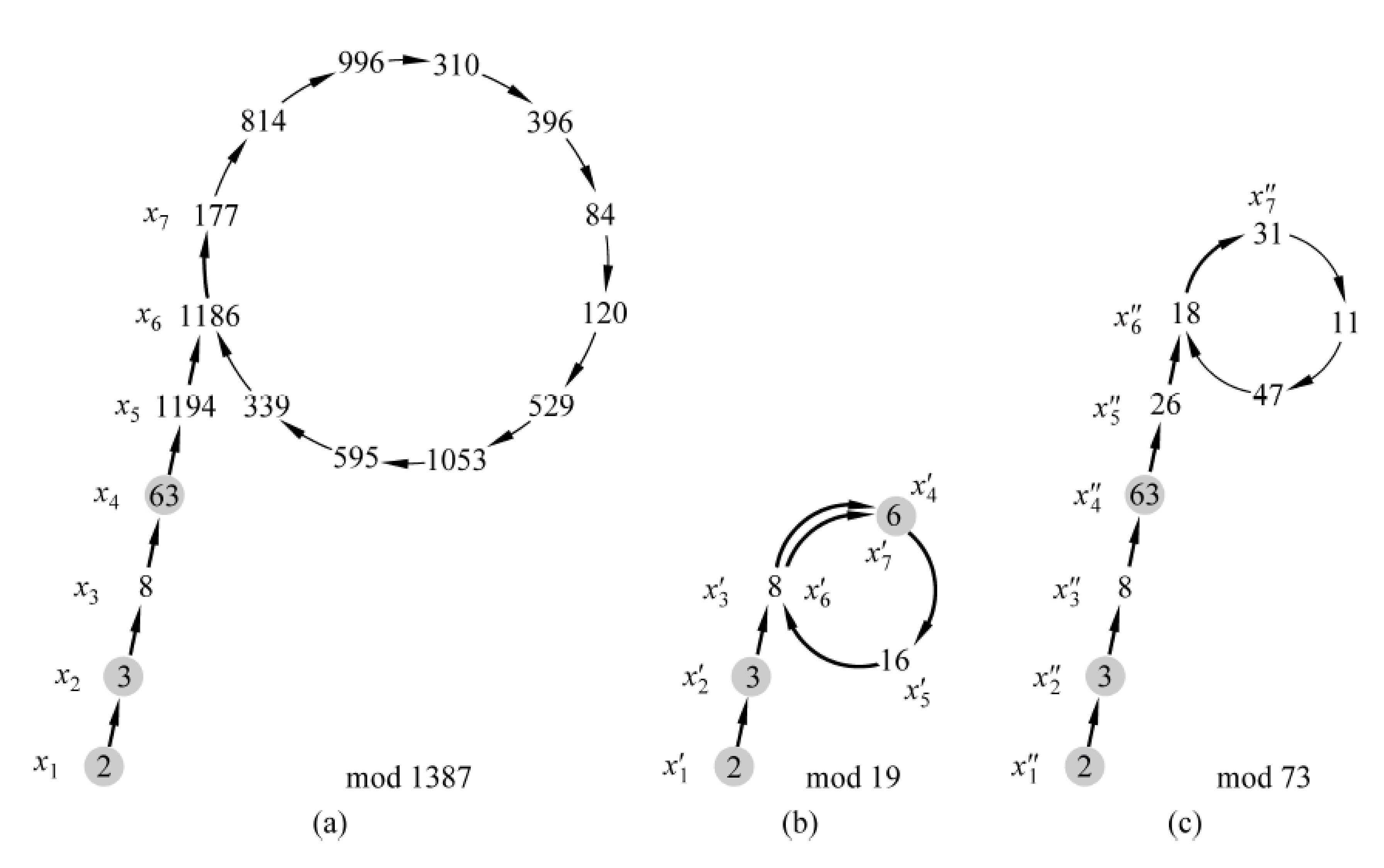


图 6-5　Pollard 的 ρ 试探

望值是 $\Theta(\sqrt{p})$。注意若 $x'_{t+i}=x'_{t+u+i}$,则 $p \mid (x_{t+u+i}-x_{t+i})$。因此,$\gcd(x_{t+u+i}-x_{t+i},\ n)>1$。

所以,一旦 POLLARD-RHO 在 y 中存储了任何 $k \geqslant t$ 的 x_k 值,则 $y \bmod p$ 总是在以 p 为模的循环周期中(若新的值存入 y,则该值也在以 p 为模的循环周期中)。最终,k 将大于 u,此时过程完成一次以 p 为模的周期循环而没有改变 y 的值。当"巧遇"事前存放的 y 时,即 $x_i \equiv y \pmod p$,就发现了 n 的一个因数。

按推测,找到的因数是素因数 p,尽管可能找到的是 p 的倍数。这是因为 t 和 u 的期望值是 $\Theta(\sqrt{p})$,产生 p 的期望步骤数是 $\Theta(\sqrt{p})$。

当然,此分析是启发式的,不严格。这是因为迭代并不是真的"随机"。无论如何,该过程在实际中运行良好,如同在此分析的那样有效。这是一个对大整数求素因数分解可选的方法。为完全分解 β 二进制位的整数 n,只需要找出所有小于$\lfloor n^{1/2} \rfloor$的素因数,我们期望 POLLARD-RHO 至多需要 $n^{1/4}=2^{\beta/4}$ 次算术运算和 $n^{1/4}\beta^2=2^{\beta/4}\beta^2$ 次位运算。POLLARD-RHO 以 $\Theta(\sqrt{p})$ 次期望的算术运算求得 n 的小素数因数 p 的能力是它最吸引人的特性。

利用素数判别过程 MILLER-RABIN 以及过程 POLLARD-RHO,可以设计出如下计算整数 n 的素因数分解的算法。

```
FACTOR(n, S)        ▷ 对 n 用 ρ 搜索法分解素因数并存储于 S
if MILLER-RABIN(n, 50)
    then S←S∪{ n }
        return
repeat
    d←POLLARD-RHO(n)
until 1<d<n
FACTOR(d, S)
FACTOR(n/d, S)
```

算法 6-19　整数素因子分解过程

假定 n 是 β 二进制位的整数，由 6.5 节知第 1 行调用 MILLER-RABIN(n, s)的时间为 $O(s\beta^3)$，第 4～6 行的 **repeat** 循环期望重复 $O(\sqrt{n})=O(2^{\beta/2})$次，每次调用 POLLARD-RHO($n$)期望耗时 $O(2^{\beta/4})$。所以，该循环期望耗时 $O(2^{3\beta/4})$。于是，若假定 d 和 n/d 的位数为 $\Theta(\beta/2)$，得到算法 FACTOR(n)运行时间的递归方程为

$$T(\beta) = O(2^{3\beta/4}) + 2T(\beta/2)$$

由此可见，用 POLLARD-RHO 分解整数素因数的时间复杂性是指数级的。

6.6.2　程序实现

1. hash 表的改造

在算法 6-18 的 POLLARD-RHO 过程中，要将伪随机序列 x_0, x_1, …，存储于集合 S 中，并在查找 n 的非平凡因子的迭代过程中随时在 S 中检测当前项 x_i 是否重复。对这样的要求，将 S 实现为 hash 表是合适的。因为在 hash 表中查找时间平均为常数。然而，直接采用第 2 章中实现的 hash 表并不能满足要求，因为存储在 hash 表中的数据是定长的整型数据，而现在要存储的是长整数 BigInt 类型。所以，需要对 hash 表做必要的改造：将对定长整型数据的操作改为对 BigInt 型数据相应的操作。把经过如此改造的 hash 表定义及其操作函数的定义及声明分别存储于 datastructure 文件夹的源文件 hash1.c 和头文件 hash1.h 中。下面仅列出数据类型的定义及向 hash 表插入操作的函数定义，读者可打开文件仔细研读。

```
1 typedef struct {                              /* hash 表结构 */
2     LinkedList ** table;                      /* 槽位数组 */
3     BigInt m;                                 /* 槽位数 */
4     size_t n;                                 /* 元素个数 */
5 } HashTable;
6 int hashInsert(HashTable * t, BigInt key) {
7     if (!inHashTable(t,key)) {                /* 若表中无 key */
8         BigInt Index=remainder(key, t->m);    /* 计算 key 的 hash 函数值 */
9         long index= *((long *)(Index.value->nil->next->key));
10        clrBigInt(&Index);
11        listPushFront(t->table[index], &key); /* 在指定槽位链表中插入 */
12        t->n++;
13        return 1;
14    }
15    return 0;                                 /* 表中已有 key */
16 }
```

程序 6-18　存储 BigInt 型数据的 hash 表定义及插入函数

对程序 6-18 的说明如下。

(1) 第 1～5 行定义了存储 BigInt 类型数据的 hash 表类型 HashTable。与 2.5.3 节的程序 2-30 的定义几乎相同，区别之一在于槽位数 m 的类型变成 BigInt。由于存储于表中的

大整数 x 的 hash 值是该数与 m 整除的余数 x mod m 决定的,为便于调用定义于源文件 bigint.c 中的函数 remainder(该函数计算两个 BigInt 型数据 a、b 相除的余数 a mod b)进行计算,所以把 m 的类型加以如此修改,尽管 m 实际大小完全可以用定长整型数据表示。其次,各槽位的链表里应存储 BigInt 型数据。

(2) 第 6～16 行定义的函数 hashInsert 完成向参数 t 指引的 hash 表中插入参数 key 指定的 BigInt 对象。若 key 能正确插入表 t 中,函数返回 1,否则(key 在 t 中已存在)函数返回 0。

(3) 函数体中,第 7～14 行处理将 key 插入 t 的操作。第 8 行调用函数 remainder 计算 key mod m,结果存储于 BigInt 型变量 Index。第 9 行将 Index 转换成 **long** 型数据 index 作为 key 的 hash 值。第 11 行和第 12 行将 key 插入到 t 的第 index 个槽位中。

2. ρ 探索法

利用经过改造的 hash 表,可将算法 6-18 实现如下。

```
BigInt PollardRho(BigInt n){
  size_t i=1,k=2;
  HashTable *S=createTable(100);
  BigInt x=random(n),y={NULL,0},d={NULL,0},t={NULL,0},tow=newIntByint(2);
  intAssign(&y,x);
  do{
        hashInsert(S,x);                                    /* S←S∪{x} */
     clrBigInt(&t);t=compareInt(&y,&x)>0? diff(y,x):diff(x,y);  /* t←|y-x| */
     clrBigInt(&d);
        d=euclid(t,n);                                      /* d←gcd(|y-x|, n) */
        if(!isOne(d)&&compareInt(d,n))                      /* d 非 n 的平凡因子 */
            break;
        i++;
        if(i==k){
            intAssign(&y,x);                                /* y←x */
            k<<=1;                                          /* k←2k */
        }
     clrBigInt(t);t=product(x,x);decrease(&t,1);
        x=remainder(t,n);                                   /* x←x^2-1 mod n */
   }while(!inHashTable(S,x));
   clrBigInt(t);clrBigInt(tow);
   clrTable(S);free(S);
   return n;
}
```

程序 6-19 实现算法 6-18 POLLARD-RHO 过程计算整数 n 的非平凡因数的函数

对程序 6-19 的说明如下。

(1) 与 POLLARD-RHO 过程相同,函数 PollardRho 只有一个表示大整数的参数 n,函数返回 n 的非平凡因数,也是大整数类型的数据。

（2）与算法过程一样，用 S 表示存储伪随机序列的集合，第 3 行调用函数 createTable(100)将其初始化为具有 100 个槽位的 hash 表。由于是迭代计算序列 xi 和 yi，所以只需要用变量 x 和 y 存储当前项。变量 d 与算法过程中的同名变量意义一致，表示 n 的非平凡因子。BigInt 型的临时变量 t 用来存储计算过程中的中间数据。tow 表示 BigInt 型的数据值 2。

（3）第 4 行调用程序 6-15 定义的函数 random(n)将 x 初始化为一个不超过 n 的随机数，第 5 行将 y 初始化为 x。第 6～20 行的 **do-while** 循环实现算法过程中第 6～16 行的 **repeat-until** 循环结构。其中，第 7 行完成 $S \leftarrow x$ 操作，第 8～10 行完成 $d \leftarrow \gcd(|y-x|, n)$ 操作。第 11 行和第 12 行处理找到 n 的非平凡因子的情形(d|n, d≠1, d≠n)。第 14～17 行完成算法过程中第12～14 行对 y 的迭代操作。注意，第 13 行用位移操作 k<<=1(等价于 k=k<<1)表示 k=2*k。第 18 行和第 19 行完成对 x 的迭代操作 $x \leftarrow x^2-1 \bmod n$。

3. 整数的素因数分解

下列函数实现算法 6-19。

```
1 void Factor(BigInt n, RBTree *s){
2   BigInt d={NULL,0},t={NULL,0},x={NULL,0};
3   if(MillerRabin(n,50)){                    /*n 为素数*/
4         intAssign(&x,n);
5         rbInsert(s, &x);                    /*n 加入解集*/
6         return;
7   }
8   do{                                       /*ρ 搜索法搜索 n 的非平凡因子 d*/
9        clrBigInt(&d);
10         d=PollardRho(n);
11   }while(!compareInt(&d,&n) || isOne(d));
12   clrBigInt(&t);t=quotient(n,d);          /*t←n/d*/
13   Factor(d);
14   Factor(t);
15   clrBigInt(&d);clrBigInt(&t);
16 }
```

程序 6-20　实现算法 6-19 FACTOR 过程的 C 函数

对程序 6-20 的说明如下。

（1）函数 Factor 与算法过程一样有两个参数：整数 n 和用来存放 n 的素因数的集合 s。之所以将 s 作为参数而不作为函数的返回值，是因为 Factor 是一个递归函数，将 s 定义成函数的局部量不合适。于是，Factor 无返回值。

（2）函数中变量 d 与算法过程中的同名变量意义一致，表示 n 的非平凡因子。t 用来表示 n 的另一个因子 n/d。由于函数向上一层返回前要释放变量 d 和 t 的存储空间，而它们是传递给下层递归的参数 n 的，其值可能在下层递归中被加入到集合 s 中，所以在加入前需要将其值复制给临时变量 x。

（3）第 3～7 行的 **if** 语句实现算法中第 1～3 行的 if 结构，调用程序 6-17 中定义的函数

MillerRabin(n,50)处理 n 为素数的情形。第 8～11 行的 **do-while** 语句实现算法过程中第 4～6 行的 **repeat-until** 结构,调用程序 6-19 定义的函数 PollardRho(n),计算 n 的非平凡因子 d。第 12 行计算 n/d,第 13 行和第 14 行对 d 和 n/d 递归处理。

为便于代码重用,程序 6-20 定义的函数存储于 numbertheory 文件夹中的源文件 factor.c 中,其原型声明于同一文件夹中的头文件 factor.h 中。文件中还存储了这两个函数的整型数据版本的定义及声明,读者可打开文件研读。

6.6.3 应用

Smith Numbers

While skimming his phone directory in 1982, Albert Wilansky, a mathematician of Lehigh University, noticed that the telephone number of his brother-in-law H. Smith had the following peculiar property: The sum of the digits of that number was equal to the sum of the digits of the prime factors of that number. Got it? Smith's telephone number was 493-7775. This number can be written as the product of its prime factors in the following way:

$$4937775=3 * 5 * 5 * 65837$$

The sum of all digits of the telephone number is 4+9+3+7+7+7+5= 42 , and the sum of the digits of its prime factors is equally 3+5+5+6+5+8+3+7= 42.

Wilansky was so amazed by his discovery that he named this kind of numbers after his brother-in-law: Smith numbers.

As this observation is also true for every prime number, Wilansky decided later that a (simple and unsophisticated) prime number is not worth being a Smith number, so he excluded them from the definition.

Wilansky published an article about Smith numbers in the Two Year College Mathematics Journal and was able to present a whole collection of different Smith numbers: For example, 9985 is a Smith number and so is 6036. However, Wilansky was not able to find a Smith number that was larger than the telephone number of his brother-in-law. It is your task to find Smith numbers that are larger than 4937775.

Input

The input consists of a sequence of positive integers, one integer per line. Each integer will have at most 8 digits. The input is terminated by a line containing the number 0.

Output

For every number $n>0$ in the input, you are to compute the smallest Smith number which is larger than n, and print it on a line by itself. You can assume that such a number exists.

Sample Input

```
4937774
0
```

Sample Output

```
4937775
```

1. 问题分析

一个合数 n 的各位数字之和等于该合数的素因子分解中各因子的数字之和，称其为 Smith 数。你的任务是编写程序，对给定的整数 a，计算大于 a 的最小 Smith 数。解决该问题是很直接的：变量 b 从 a 起逐一检测，若 b 是合数计算 b 的各位数字和 s1，然后分解 b 的素因子，计算所有因子的个位数字之和 s2，若 s1=s2，则输出 b，否则对 b+1 做相同的检测。

2. 程序实现

将上述的解题思想实现为如下程序。

```
int s2;                                  /* 保存各素因子的数字之和的全局变量 */
int sumdigit(long n){                    /* 计算整数 n 的各位数字之和 */
  int s=0,m=n;
  do{
      s+=m%10;
      m/=10;
  }while(m);
  return s;
}
void sumdigits(unsigned long *a){
   s2+=sumdigit(*a);
}
int main(){
   int a;
   FILE *f1=fopen("chap06/SmithNumbers/inputdata.txt","r"),
        *f2=fopen("chap06/SmithNumbers/outputdata.txt","w");
   assert(f1&&f2);
   fscanf(f1,"%d", &a);
   while(a){                             /* 对输入文件中的每一个案例数据 a */
       unsigned long s1,b=a;
       while(1){                         /* b 从 a 起逐一检测 */
          RBTree *factors=creatRBTree(sizeof(unsigned long),unsignedGreater);
          if(isPrime(b)){                /* 排除 b 为素数 */
              b++;
              continue;
          }
          s1=sumdigit(b);                /* 计算 b 的各位数字之和 */
```

```
28             s2=0;
29             factor(b,factors);          /* 对 b 做素因子分解,结果放在 factors 中 */
30             inorderRBWalk(factors,sumdigits);clrRBTree(factors,NULL);
31             if(s1==s2){                 /* 满足 Smith 数条件 */
32                 fprintf(f2,"%d\n",b);
33                 break;
34             }
35             b++;
36         }
37         fscanf(f1,"%d",&a);
38     }
39     fclose(f1);fclose(f2);
40     return 0;
41 }
```

程序 6-21 解决 Smith Numbers 问题的 C 程序

对程序 6-21 的说明如下。

(1) 第 1 行定义的整型全局变量 s2 用来存储整数 b 的所有素因子各位数字之和。

(2) 第 2～9 行定义的函数 sumdigit 计算整型参数 n 的各位数字之和,并作为函数值返回。第 10～12 行定义的函数 sumdigits 将由指针参数 n 指引的整数的各位数字之和累加到全局变量 s2 中。

(3) main 函数中的第 19～38 行的 **while** 循环处理输入文件 f1 中的每一个案例数据 a。其中,内嵌于第 21～36 行的 **while** 循环对 b 从 a 开始逐一检测,直至得到不小于 a 的最小 Smith 数。对每一个合数 b,第 27 行调用函数 sumdigit 计算其各位数字之和,保存在 s1 中。第 29 行调用函数 factor,计算 b 的素因子分解,各因子存储于 RBTree 类型指针 factors 指引的树结构中。该函数是程序 6-16 的整型版本。第 30 行调用程序 2-17 中定义的函数 inorderRBWalk,对二叉树结构 factors 作中序遍历,利用函数 sumdigits 将存储在 factors 中的 b 的各因子数字和累加到 s2 中。注意,对每一个合数 b,第 28 行将 s2 初始化为 0。若 s1 与 s2 相等,第 32 行将 b 写入输入文件 f2,否则检测下一个 b。

第 7 章　回 溯 策 略

人类的活动离不开资源竞争，在占有有限资源的前提下，最优化活动的目标是人们的普遍追求。这就引出了一类问题：如何才能在有限资源的约束条件下最优化活动目标？由于在约束条件下，可以有多种达到目标的方法或途径，这类问题往往呈现出具有多个可能解，每一个解都对应一个目标值，目的是找出目标值最优的解这样的特征。人们将具有这一特征的问题称为**组合优化问题**。例如，下面这些问题就涉及最优化组合。

（1）航空公司希望调度其航班人员。航空管理局有很多强制性的约束，如航班成员连续工作的时数，一个航班成员在一个月内只能工作于一种型号的飞机上。航空公司希望用尽可能少的航班成员来调度所有的航班。

（2）石油公司希望确定在哪里打井。钻井的设置与成本、地质勘测和每桶油的平均工资支付相关。公司对钻井的选址预算是有限的，对给定的预算希望得到最大量的石油。

用人力来解组合优化问题往往因为计算量太大而可望不可及。自从有了计算机，快速地解决所有的组合优化问题似乎看到曙光。于是大量的计算机科学家投入到了设计出解决组合优化问题的有效算法的攻关中了。事情远不是人们开始时所期盼的那样乐观，甚至到了今天，人们仍然离彻底解决这个问题非常遥远。但是，透过重重迷雾，人们已经有理由乐观起来：虽然只对一部分组合优化问题找到了有效算法（运行时间上界是输入规模 n 的多项式），对组合优化问题作深入的分类研究发现其中一大部分问题——称为 **NP 完全问题**——具有共同的性质，只要能找到其中一个问题的有效算法，则所有 NP 完全问题都有有效算法。一旦说明其中之一不存在有效算法，则所有的 NP 完全问题都不存在有效算法了。从本章起，将用 3 章的篇幅讨论解决组合优化问题的各种算法。

7.1　组合问题

在解决组合优化问题之前，先来看稍微简单一点的组合问题，通过几个例子来引入这一概念。

7.1.1　组合问题的例子

1. *m*-着色问题

考虑 3-着色问题：给定一个无向图 $G=\langle V, E\rangle$，其中，顶点集 $V=\{1, 2, \cdots, n\}$，边集 $E\subseteq V\times V$。需要对其中的每一个顶点着 1、2、3 三种颜色之一，使得任意两个相邻顶点颜色不同。人们把这样的一种着色方案称为是**合法**的；反之，若两个相邻顶点具有同种颜色，则是**不合法**的。一个着色方案可表示为一个 n 维向量 $\langle x_1, x_2, \cdots, x_n\rangle$，其中 $x_i\in\{1, 2,$

3},表示第 i 个顶点的着色,$1\leqslant i\leqslant n$。例如,<1, 2, 2>表示一个具有 3 个顶点的图的着色方案。对一个具有 n 个顶点的图 G 而言,有 3^n 种可能的着色方案(包括合法的与不合法的)。一个问题的所有可能解构成的集合称为该问题的**解空间**,常用符号 Ω 表示。例如,对一个具有 3 个顶点的图,3-着色问题的解空间 Ω 如图 7-1 所示。

图 7-1　一个具有 3 个顶点的图的所有可能解

一共有 $3^3=27$ 个方案。m-色问题可形式化地描述如下。

输入:图 $G=<V, E>$。其中,$V=\{1, 2, \cdots, n\}$,$E\subseteq V\times V$。颜色数为 m。

输出:若 G 存在的各顶点的合法着色方案,输出所有解向量 $x=<x_1, x_2, \cdots, x_n>$,其中 $1\leqslant x_i\leqslant m$ 为 G 的第 i 个顶点的着色,且对 $\forall(i, j)\in G[E](x_i\neq x_j)$[①]。

用穷尽搜索法在解空间中求得问题的解的算法称为强力算法。对每个方案需要检测每个顶点的着色,对具有 n 个顶点的图来说,在具有 m^n 个可能解的解空间 Ω 中搜索可行解的运行时间为 $\Omega(m^n)$。

2. 子集和问题

子集和问题是在一个整数集合 $A=\{a_1, a_2, \cdots, a_n\}$中,寻求一个子集合 $A'\subseteq A$,使得 A'中元素之和等于给定的整数 C。例如,整数集合 $A=\{1, 2, 3, 4\}$,整数值 $C=4$,则 A 的子集{4},{1, 3}都是合法解。

用向量 $x=<x_1, x_2, \cdots, x_n>$表示子集和问题的解。其中 $x_i=\begin{cases}1 & a_i\text{ 被选中}\\0 & a_i\text{ 未被选中}\end{cases}$,$i=1, 2, \cdots, n$。于是,对上述整数集合 $A=\{1, 2, 3, 4\}$,整数值 $C=4$ 的子集和问题的两个合法解的向量表示为<0, 0, 0, 1>和<1, 0, 1>。子集和问题的形式化描述如下。

输入:整数集合 $A=\{a_1, a_2, \cdots, a_n\}$,整数值 C。

输出:向量 $x=<x_1, x_2, \cdots, x_k>$,其中 $x_i\in\{0, 1\}$,若 $a_i\in A$,$x_i=1$,否则 $x_i=0$,$i=1, 2, \cdots, k(\leqslant n)$。使得 $\sum_{i=1}^{k} x_i\cdot a_i=C$。

子集和问题的可能解是 n 维向量$<x_1, x_2, \cdots, x_n>$,其中 $x_i\in\{0, 1\}$,$i=1, 2, \cdots, n$。因此,子集和问题的解空间 Ω 含有 2^n 个向量。对所有的可能解逐一检测,在解空间中搜寻所有合法解的强力算法的运行时间下界为 $\Omega(2^n)$。

3. n-皇后问题

经典的 8-皇后问题可阐述如下。如何在一个 8×8 的棋盘上放置 8 个皇后使得任意两个皇后之间不能相互攻击?若两个皇后同处于一行,一列或一斜线上,则它们可相互攻击。n-皇后问题的定义是类似的,即对任意的 $n\geqslant 1$,有 n 个皇后及一个 $n\times n$ 的棋盘。

例如,当 $n=4$ 时,考虑一个 4×4 的棋盘。由于不能将两个皇后置于同一行中,每一行

① ∀是数理逻辑中的全称代词,意为"对所有的……"。

只能有一个皇后。每一行中有 4 个位置，总共有 4^4 种可能的格局。每一种格局可以用一个具有 4 个分量的向量 $x=\langle x_1, x_2, x_3, x_4\rangle$ 加以表示，其中 x_i 表示棋盘第 i 行的皇后放置位置。例如，向量 $\langle 2, 3, 4, 1\rangle$ 对应于展示在图 7-2 中的格局。若在对应的行中还没有放置皇后，则该分量为零。事实上，由于不能将两个皇后置于同一列，任何一个放置方案都应对应于 1, 2, 3, 4 的一个排列。

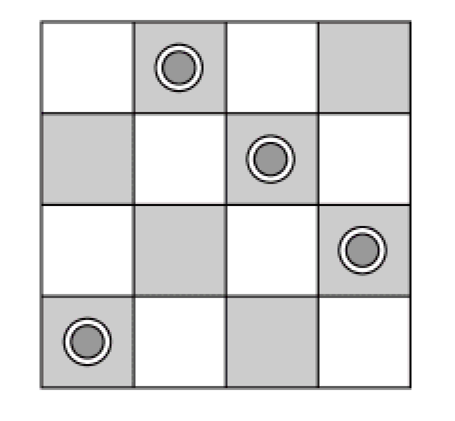


图 7-2　4-皇后问题的一个格局

将 n-皇后问题形式化为如下。

输入：棋盘规模 n。

输出：如果此 $n\times n$ 棋盘存在合法格局，则输出所有由向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$ 组成的解，其中，$x_1, x_2, \cdots, x_n$ 是 $1, 2, \cdots, n$ 的一个排列，且 $\forall(1\leqslant i\neq j\leqslant n)$ 有 $x_i - x_j\neq i-j$ 且 $x_i-x_j\neq j-i$ 决定的格局：棋盘上的第 i 行第 x_i 格放置一个皇后，$i=1, 2, \cdots, n$。否则，输出无解信息。

n-皇后问题的可能解为 n 维向量 $\langle x_1, x_2, \cdots, x_n\rangle$，其中，$x_1, x_2, \cdots, x_n$ 是 $1, 2, \cdots, n$ 的一个排列。所以，所有可能解构成的解空间 Ω 含有 n^n 个元素。对每个可能解检测其合法性，在解空间中搜索合法解的强力算法将耗时 $\Omega(n^n)$。

7.1.2　组合问题的形式化描述

考虑上述 3 个问题，它们有一些相同的特征。首先，它们的解都可表示为 n 维向量 $\langle x_1, x_2, \cdots, x_n\rangle$。每个分量 x_i 取值于一个有限集合 X_i，$i=1, 2, \cdots, n$。在 3-着色问题中，$X_i=\{1, 2, 3\}$，$i=1, 2, \cdots, n$。子集和问题中，$X_i=\{0, 1\}$，$i=1, 2, \cdots, n$。而在 n-皇后问题中，$X_i=\{1, 2, \cdots, n\}$，$i=1, 2, \cdots, n$。其次，有一个判断向量 $x=\langle x_1, x_2, \cdots, x_n\rangle\in\Omega=X_1\times X_2\times\cdots\times X_n$ 是否合法的函数 f：$\Omega\rightarrow\{\text{true}, \text{false}\}$。在 3-着色问题中，此 f 就是检测图 G 中任意两个顶点 $i, j\in V[G]$，若边 $(i, j)\in E[G]$，x_i 是否不等于 x_j。在子集和问题中，此 f 就是检测 $\sum_{i=1}^{n} x_i\cdot a_i$ 是否等于给定的值 C。而在 n-皇后问题中，此 f 就是检测每个 x_i 与其他 $n-1$ 个分量是否不等且不在一条直线上。

我们的任务是在所有的这样的向量构成的解空间 Ω 中搜索合乎条件 $f(x_1, x_2, \cdots, x_n)=\text{true}$ 的合法解。于是可以一般地将组合问题描形式化地述为如下。

输入：n 个有限集合 $X_1, X_2, \cdots, X_n$ 构成的笛卡儿积 $\Omega=X_1\times X_2\times\cdots\times X_n$。

输出：Ω 的子集 S，使得 $\forall\langle x_1, x_2, \cdots, x_n\rangle\in S, f(x_1, x_2, \cdots, x_n)=\text{true}$。

对于组合问题，可以用如下强力算法解决。

```
BRUTE-FORCE(X1, X2, …, Xn)
Ω←X1×X2,×…×Xn
for each x=<x1, x2, …, xn>∈Ω
    do if f(x1, x2, …, xn)=true
        then print x
```

算法 7-1　解决组合问题的强力算法

设 $|\Omega|=g(n)$，f 的运行时间为 $t(n)$，则 BRUTE-FORCE 的运行时间是 $T(n)=\Theta(g(n)\cdot t(n))$。例如，3-着色问题中 $g(n)=3^n$，由于对每个可能解需要检测 n 个分量是否合法，故 $t(n)=n$，BRUTE-FORCE 解决 3-着色问题的运行时间 $T(n)=\Theta(n3^n)$。类似地，子集和问题中，$g(n)=2^n$，$t(n)=n$，BRUTE-FORCE 解决子集和问题的运行时间 $T(n)=\Theta(n2^n)$。而在 n-皇后问题中，$g(n)=n^n$，$t(n)=n$，BRUTE-FORCE 解决 n-皇后问题的运行时间 $T(n)=\Theta(n^{n+1})$。

7.2 组合问题的回溯算法

对很多组合问题运用强力算法 BRUTE-FORCE 解决，会遭遇时间黑洞。因为很多组合问题的解空间 Ω 中可能解的个数 $g(n)$($g(n)=|\Omega|$)是输入规模 n 的指数表达式，例如，上述 3 个例子都是这样的情况。于是人们把注意力放在如何压缩解空间，来提高算法的运行效率。一个粗略的想法是按照某种层次结构组织解空间，虽然这样做并没有减小解空间，但能够减少搜索合法解时所做的检测次数。

7.2.1 解空间的树状结构

仍以上述 3 个问题为例来说明解空间的树状结构。

1. *m*-着色问题

由 7.1 节的讨论，我们知道具有 n 个顶点的图 G 的 3-着色问题的解空间 Ω 包含 3^n 个可能解，为判断 Ω 中的一个形式为 n 维向量的可能解是否合法，需要检测其所有 n 个分量，因此，解决 3-着色问题的强力算法 BRUTE-FORCE 要做 $n3^n$ 次检测。图 7-1 展示了具有 3 个顶点的图 G 的 3-着色问题的解空间。仔细考察其中的 27 个着色方案，可以发现很多方案具有相同的部分：前 9 个方案的第一个顶点的都着 1 色，前 3 个方案的前两个顶点都着 1 色，……。因此，考虑把前面相应分量值相等解组合起来，将解空间组织成一个树结构。例如，对具有 3 个顶点的图，其解空间可表示为图 7-3 所示的称为**搜索树**的完全三叉树。

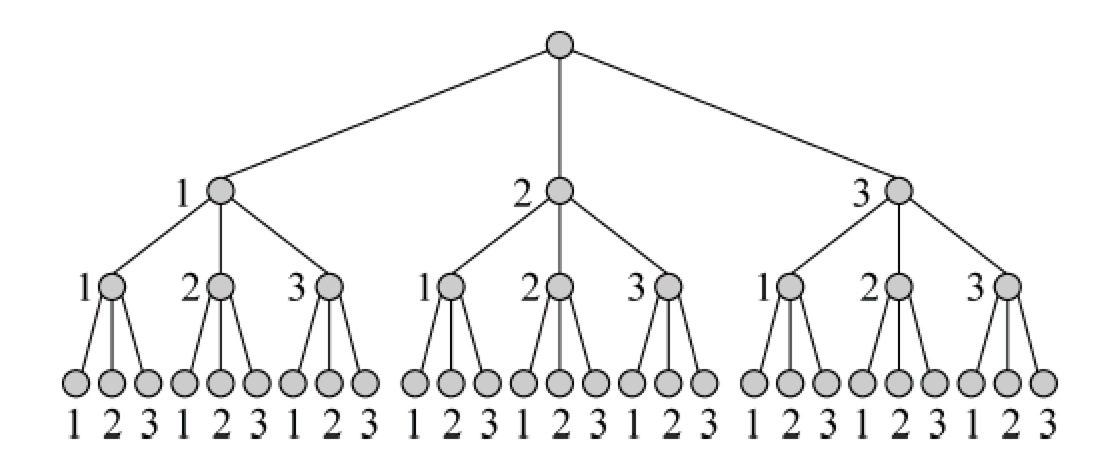


图 7-3 一个具有 3 个顶点的图 G 的 3-着色问题解空间的树结构

在搜索树中，树根表示初始状态——尚未考虑解向量的任何分量。树根下的第 1 层结点表示解向量 x 的第 1 个分量所有可能的取值。在 3-着色问题中，第 1 个分量有 3 种可能的着色方式：1 或 2 或 3。对应树根下第 1 层的 3 个结点。对于第一个分量每一种着色方

式，第 2 个分量也有 3 种着色方式。所以树根下第 2 层共有 9 个结点，它们分别是第 1 层的 3 个结点的孩子。同理，根下第 i 层的 3^i 个结点对应解向量中第 i 个分量在第 $i-1$ 个分量确定后的各种着色方式。于是，对解空间中的 3^n 个可能解的每一个，对应搜索树中从根开始到达一片叶子的路径上除根以外的各个结点组成的向量。

于是，对解空间中所有可能解判断合法性过程中检测的次数为

$$\begin{aligned}3+3^2+\cdots+3^n &= 1+3+3^2+\cdots+3^n-1\\ &=(3^{n+1}-1)/2-1\\ &<3^{n+1}/2\\ &\leqslant n3^n/2(n\geqslant 3)\end{aligned}$$

即当 $n\geqslant 3$ 时，对组织成搜索树的解空间中可能解判断合法性对向量的分量所做的检测次数比对原本的解空间做同样操作时所作的检测次数减少了一大半。

2. 子集和问题

由 7.1 节知，对具有 n 个整数 $a_1, a_2, \cdots, a_n$，以及为 C 的子集和问题的每个可能解 x 是一个 n 维向量，其每一个分量非 0 即 1。故解空间 Ω 包含 2^n 个可能解。为检测可能解的合法性需检测其每一个分量。于是，解决子集和问题强力算法 BRUTE-FORCE 要做 $n2^n$ 次检测。仿照 3-着色问题，可将此 2^n 个可能解组织成一棵搜索树。由于解向量的每个分量取且仅取 0 和 1 两值之一，故子集和问题的搜索树是一棵完全二叉树。图 7-4 展示了 $n=4$ 时的搜索树。

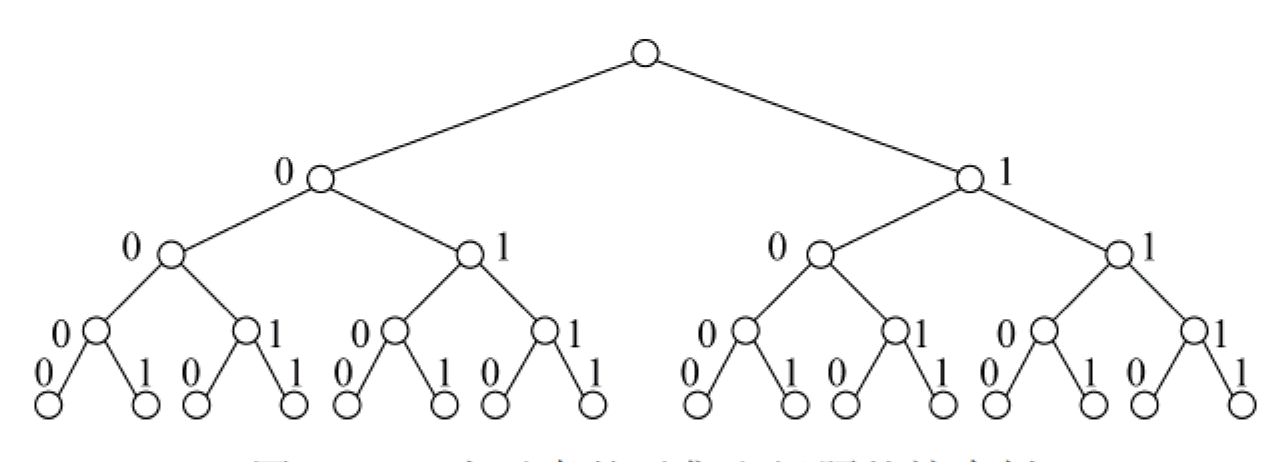


图 7-4　4 个元素的子集和问题的搜索树

将子集和问题的解空间 Ω 组织成搜索树后，每一个可能解对应树中从根开始到达一片叶子的路径上除了根结点以外的 n 个结点构成的向量。

此时对解空间中的可能解判断其是否合法所做的检测次数为

$$\begin{aligned}2+2^2+\cdots+2^n &= 1+2+2^2+\cdots+2^n-1\\ &=2^{n+1}-2\end{aligned}$$

而对原本的解空间做同样操作时所做的检测次数 $n2^n$，两者相差

$$\begin{aligned}n2^n-(2^{n+1}-2) &= n2^n-2^{n+1}+2\\ &=2^n(n-2)+2\end{aligned}$$

也就是说，后者比前者少做了 $2^n(n-2)+2$ 次检测。

3. n-皇后问题

对棋盘规模为 $n\times n$ 的 n-皇后问题，其解空间 Ω 含有 n^n 个可能解，对每个表示为 n 维向量的可能解为判断其是否合法，必须检测其每一个分量。故解决 n-皇后问题强力算法

BRUTE-FORCE 要做 n^{n+1} 次检测。与 3-着色问题和子集问题相仿,可以把解空间组织成一棵结构为 n 叉完全树的搜索树,使得 Ω 中的每一个可能解对应搜索树中从根开始到达一片叶子的路径上除了根结点以外的 n 个结点构成的向量。图 7-5 展示了 4-皇后问题组织成搜索树的解空间。其中根下第 1 层结点对应第 1 行中皇后的摆放位置,由于图形比较繁杂,故仅列出第 1 棵和第 4 棵子树,省略了第 2 棵、第 3 棵子树。

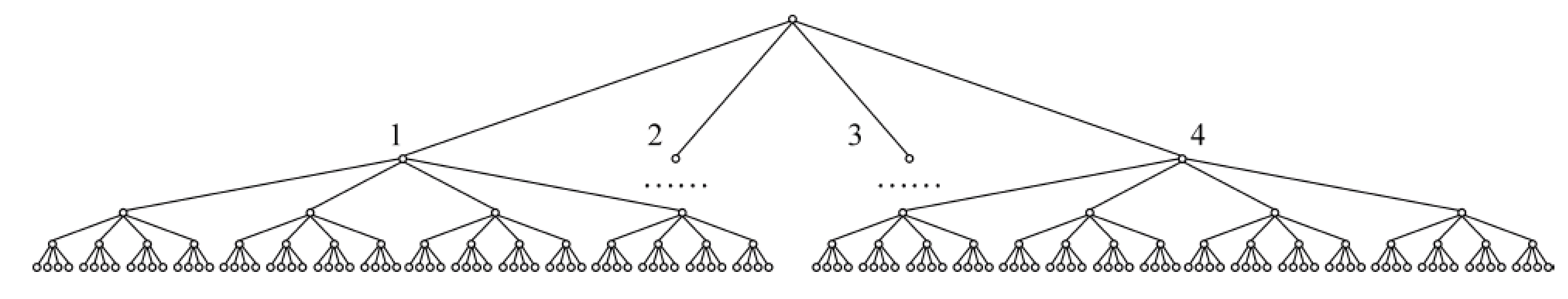


图 7-5　4-皇后问题的搜索树

对组织成搜索树的解空间中的所有可能解判断其合法性时,对解向量中各分量的检测次数为

$$
\begin{aligned}
n+n^2+\cdots+n^n &= 1+n+n^2+\cdots+n^n-1\\
&= \frac{n^{n+1}-1}{n-1}-1\\
&= \frac{n^{n+1}-n}{n-1}\\
&\leqslant \frac{n^{n+1}-n}{n/2}\\
&= 2n^n-2
\end{aligned}
$$

对原本的解空间做同样操作时所做的检测次数相比,两者之差为

$$
\begin{aligned}
n^{n+1}-\frac{n^{n+1}-n}{n-1} &\geqslant n^{n+1}-2n^n+2\\
&> n^{n+1}-2n^n\\
&= n^n(n-2)
\end{aligned}
$$

也就是说,解空间组织成搜索树后,检测次数将至少减少 $n^n(n-2)$次。

7.2.2　解决组合问题的回溯算法

我们已经看到,将解空间组织成搜索树后,在解空间中搜索合法解时,可能减少大量的检测时间。但须对强力算法 BRUTE-FORCE 过程加以修改,使其能在搜索树中搜索所有的合法解。仍然以上述的 3 个组合问题为例子。

1. m-着色问题

搜索过程维护一个渐增的**部分解**:已合法着色的部分顶点。利用搜索树,从树根下第一层结点起,按**深度优先**的策略进行搜索:对当前结点着色,判断其是否合法。若是,则进入下一层结点进行搜索,否则变换到当前结点的兄弟结点,即淘汰本结点及其子孙结点。若

对应的顶点对 3 种着色均不是合法的，则搜索回溯到当前结点的父结点，并变换到其兄弟结点。若从根起到达树中的一片叶子的着色方案是合法的，输出合法解，并回溯。

例如，考虑图 7-6(a)所示的图 G，要对其顶点用颜色{1，2，3}来着色。图 7-6(b)部分展示了合法着色方案形成过程中搜索树的路径。首先，在产生出第 3 个结点后，发现着色方案<1，1>不是合法部分解，所以该结点在图中用“×”标识为死结点。接着 b 涂成颜色 2，这样着色方案<1，2>看起来是一个部分解。于是产生一个对应于顶点 b 的新的孩子结点，其初始着色为 1。重复上述过程，最终将得到一个合法的着色方案<1，2，2，1，3>。由于搜索树枝繁叶茂，一页画面不足以展示整个搜索过程，仅画出 $a=1$ 的子树部分。读者可自行补充其余部分。

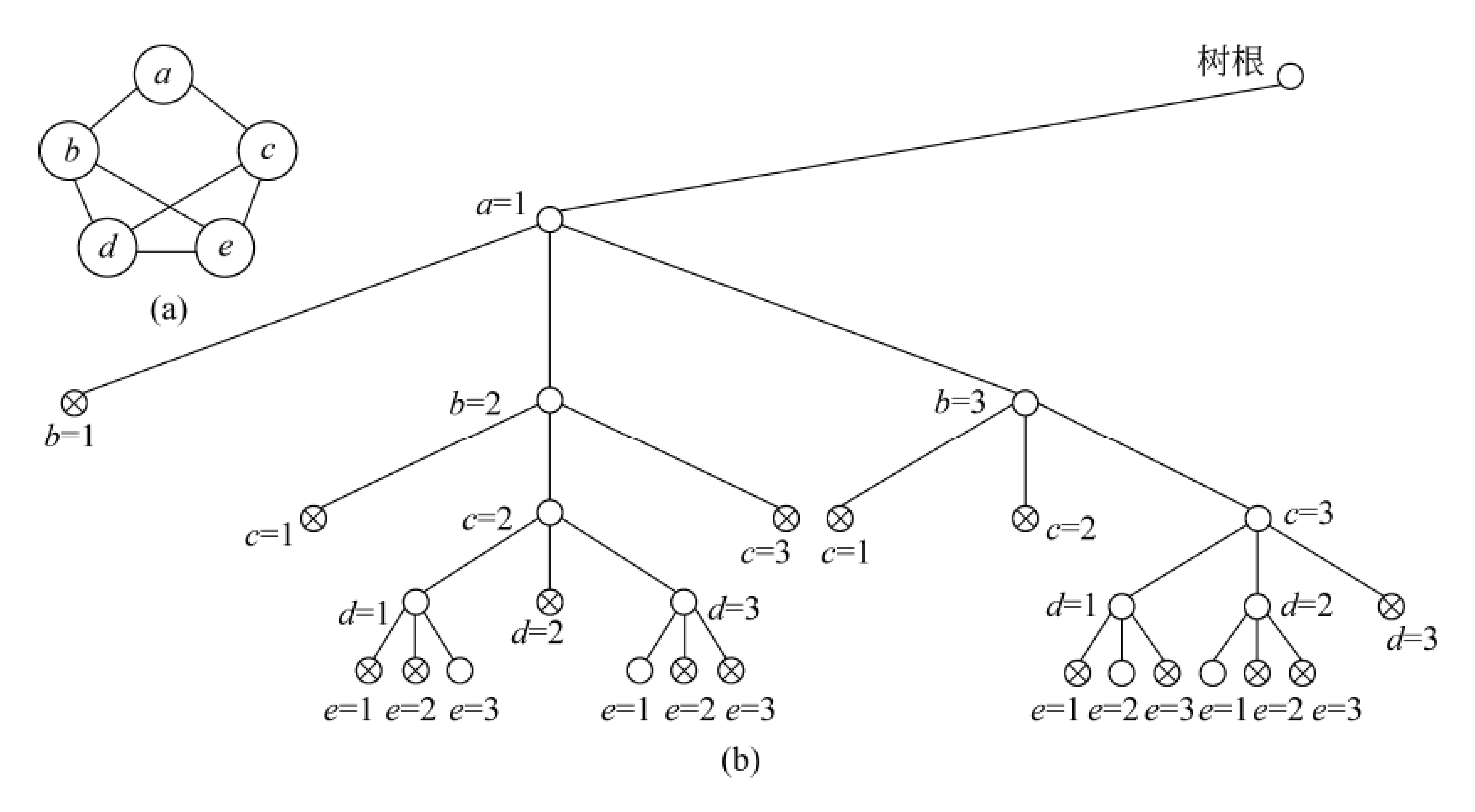


图 7-6　利用回溯解决 3-着色问题的一个例子

上述的想法用伪代码表示为过程 M-COLOR 和 GRAPH-COLOR。

```
M-COLOR(G)
1 solutions←∅
2 为解向量 x 分配存储空间
3 GRAPH-COLOR(G, 1)          ▷从根起对以下第 1 层结点进行搜索
4 return solutions
```

算法 7-2　求解 3-着色问题的算法

算法 7-2 中第 3 行调用下列递归算法，以回溯策略探索能否对图 G 进行 3-着色。

```
GRAPH-COLOR(k)
if k>n                                          ▷判断是否为完整解
  then solutions← solutions ∪{<x1, x2, …, xn>}
    return
for color←1 to m                                ▷对当前第 k 个顶点逐一检测 3 种可能的着色
  do xk←color
    if ∀1≤i≤k((i, k)∈E[G] →xi≠ xk)①            ▷部分合法
```

① 此处符号→是数理逻辑中的蕴涵连接词，对两个命题 a 与 b，命题 a→b 意为“若 a 则 b”。

```
7        then GRAPH-COLOR(k+1)                 ▷进入下一层搜索
```

算法 7-3 探索图 G 的第 k 个顶点的着色方案的递归过程

对算法7-2和算法7-3的说明如下。

(1) 算法7-3是在前 $k-1$ 个顶点具有合法着色的前提条件下进行计算的。第1～4行检测是否 $k>n$,若是意味着 x 已是一个完整的合法解,将其加入到合法解集合 *solutions* 中。

(2) 若 $k\leqslant n$,第4～7行的 **for** 循环对第 k 个顶点逐一检测 m 种着色:若检测到一种颜色是合法的,则对第 $k+1$ 个顶点递归地进行探索。若3种着色都不合法将回溯到上一个顶点继续检测其他可能的着色。

(3) 应当注意的是,无须存储整棵搜索树,只需要存储从根到当前活动结点的路径。事实上,根本就不产生实际的结点,整棵树是隐含的。仅需要跟踪已着颜色的各个顶点(也就是解向量 x 的前 k 个分量)。

上述的递归过程易于理解,但对计算机而言,递归过程比迭代过程消耗更多的系统资源(系统栈)。注意到算法7-3的递归调用处于过程的末尾,对于末尾递归的算法很容易转换成与之等价的迭代算法。

```
m-COLOR(G, m)
1 solutions←∅
2 allocate x and for 1≤i≤n set x_i as 0
3 k←1                                         ▷从第1层开始探索
4 while k≥1
5     do while x_k<m                           ▷x_k∈X
6         do x_k←x_k+1
7            if ∀1≤i≤k((i, k)∈E[G] →x_i≠ x_k)  ▷部分合法
8              then if k=n                     ▷完整解,输出
9                     then solutions← solutions ∪{<x_1, x_2, …, x_n>}
10                         exit this while loop
11                    else k←k+1               ▷合法但不完整,进一步探索
12    x_k←0
13    k←k-1                                    ▷回溯
14 return solutions
```

算法 7-4 解决 m-着色问题的回溯算法的迭代版本

算法7-4是算法7-2与算法7-3的结合。第1～2行完成与算法7-2的第1行和第2行操作一样的任务,初始化合法解集合 *solutions* 并为解向量 x 分配存储空间。稍有不同的是,在迭代版本中,解向量的各分量是通过自身增值来改变颜色号(见算法7-4的第6行),而在递归版本中是对其直接赋值来改变的(见算法7-3的第5行)。所以,算法7-4中,在为 x 分配空间的同时,还需将其所有分量 x_i 初始化为0。算法7-4的第14行对应算法7-2的第4行,即整个解空间搜索完毕,返回合法解集合 *solutions*。算法7-4的第3～13行等价于算法7-2中第3行对算法7-3的调用 GRAPH-COLOR(G, 1)。由于 GRAPH-COLOR 是一个末尾递归的过程,所以这一部分可用一个 **while** 循环(第4～13行)来替代递归。需要注意的是,此处第5～11行的 **while** 循环与算法7-3中第4～7行的 **for** 循环是有区别的。此处的

循环条件是 $x_k<m$，而 k 在循环体中可能发生变化（见第 11 行向下层进一步探索），所以 x_k 本次的值是根据前次值自增得以变化。此外，该 **while** 循环结束意味着对当前的 x_k 检测完了所有可能的值，需要回溯，回溯前 x_k 要清 0（第 12 行）。而对于算法 7-3 的第 4～7 行的 **for** 循环而言，循环中 k 不会变化，向下层探索的任务是由递归来完成的。于是无须记忆 x_k，这一工作由系统栈来完成。

鉴于迭代算法比递归算法的执行效率更高，本书今后均以迭代方式描述回溯算法。

2. 子集和问题

与解决 3-着色问题类似的搜索思路，对子集和问题的搜索树，从树根下第 1 层开始，按深度优先策略在树中搜索合法解。例如，对整数集合 $A=\{1, 2, 3, 4\}$，整数值 $C=4$，且以 $\sum_{i=1}^{k} x_i a_i \leqslant C$ 为判断 $(x_1, x_2, \cdots, x_k)$ 是否部分合法的规则，图 7-7 展示了在搜索树中搜索合法解的过程，图中的死结点标为“×”。

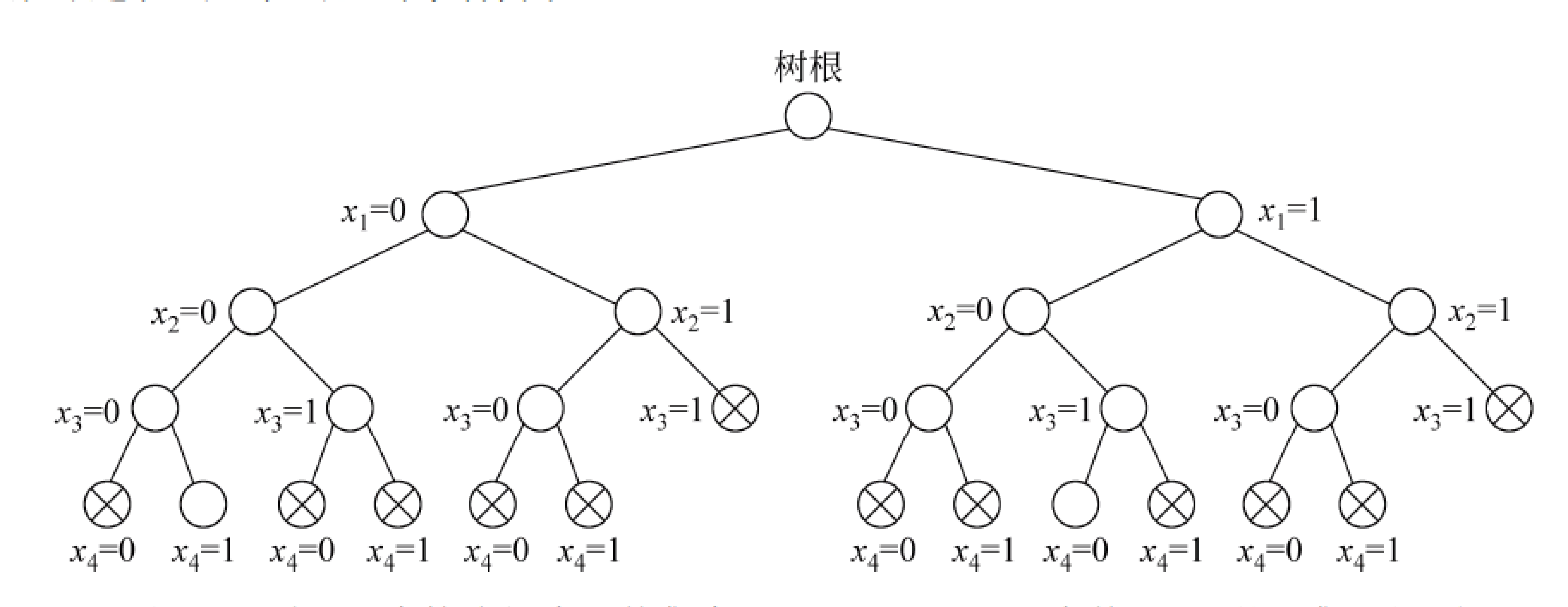


图 7-7　利用回溯策略解决整数集合 $A=\{1, 2, 3, 4\}$，常数 $C=4$ 的子集和问题

搜索过程从根下第一层 $k=1$ 的结点 $x_1=0$ 开始，只要 $\sum_{i=1}^{k} x_i a_i \leqslant 4$，就往下一层继续探索，这样首先会探索到部分解 $\langle x_1, x_2, x_3\rangle=\langle 0, 0, 0\rangle$ 是合法的部分解。继续往下一层探索 $\langle x_1, x_2, x_3, x_4\rangle=\langle 0, 0, 0, 0\rangle$ 虽然部分合法，但 $n=4$ 已经无法继续往下探索，故 $\langle x_1, x_2, x_3\rangle=\langle 0, 0, 0\rangle$ 时，$x_4=0$ 是一个死结点。x_4 自增 1 后成为 1，此时 $\langle x_1, x_2, x_3, x_4\rangle=\langle 0, 0, 0, 1\rangle$ 构成一个合法解，输出后回溯。图中其他死结点的判定及另一合法解 $\langle x_1, x_2, x_3, x_4\rangle=\langle 1, 0, 1, 0\rangle$ 的获得与此相似，不再赘述。将回溯过程写成伪代码如下。

```
SUBSET-SUM(A)
solutions ← ∅
allocate x and set each x_i as −1
k ← 1                                         ▷从第 1 层开始探索
while k ≥ 1
    do while x_k < 1
        do x_k ← x_k + 1                      ▷x_k ∈ X = {0,1}
            if Σ_{i=1}^{k} x_i a_i ≤ C         ▷部分合法
```

```
8               then if ∑_{i=1}^{k} x_i a_i = C           ▷合法解,输出
9                   then solutions ← solutions ∪{< x_1, x_2, …, x_k >}
10                       exit this while loop
11                  else if k≥n                           ▷达到树叶,但并非合法
12                      then if x_k≥1
13                              then exit this while loop
14                      else k←k+1                        ▷合法但不完整,进一步探索
15      x_k←−1
16      k←k−1                                             ▷回溯
17 retun solutions
```

算法 7-5　解决子集和问题的回溯算法

由于解向量中每个分量的取值范围均为{0, 1},故第 2 行将所有分量初始化为−1,第 15 行回溯前将 x_k 置为−1,以呼应第 4～12 行的 x_k 的自增 1 操作。

3. *n*-皇后问题

在 n-皇后问题组织成搜索树中按回溯策略搜索合法解的过程与前两个例子是一样的。对于 $n=4$ 的情形,算法找到一个合法解的执行过程如图 7-8 所示。图中的死结点标为“×”。首先,x_1 置为 1 且 x_2 置为 1。这将导致一个死结点,因为这两个皇后置于同一列中。若 x_2 置为 2 也将发生同样的结果,因为这两个皇后置于同一斜线上。将 x_2 置为 3 将导致一个部分解向量<1, 3>且探索对 x_3 寻求一个值。如在图 7-8 中所示,无论 x_3 假设为何值,都不会得到 $x_1=1$, $x_2=3$ 及 $x_3>0$ 的部分解。所以搜索回溯到第二层并对 x_2 赋予一个新的值,即 4。如图所示,这导致了部分解向量<1, 4, 2>。此向量还是不能被扩展,在产生出一些新的结点后,搜索回到第一层。现在,x_1 增加为 2,以同样的方式,找到部分解向量<2, 4, 1>。如此,此向量被扩展为合法向量<2, 4, 1, 3>,这就对应一个合法解。搜索过程写成伪代码如下。

```
N-QUEENS(n)
solutions←∅
k←1                                                  ▷从第 1 层开始探索
allocate x and set each x_k as 0
while k≥1
    do while x_k<n                                   ▷x_k∈X
        do x_k←x_k+1
            if ∀ 1≤i<k(x_i≠ x_k and |x_i−x_k|≠|i−k|)  ▷合法部分解
                then if k=n                          ▷完整解,输出
                    then solutions ← solutions ∪{< x_1, x_2, …, x_n >}
                        exit this while loop
                    else k←k+1                       ▷合法但不完整,进一步探索
    x_k←0
    k←k−1                                            ▷回溯
return solutions
```

算法 7-6　解决 n-皇后问题的回溯算法

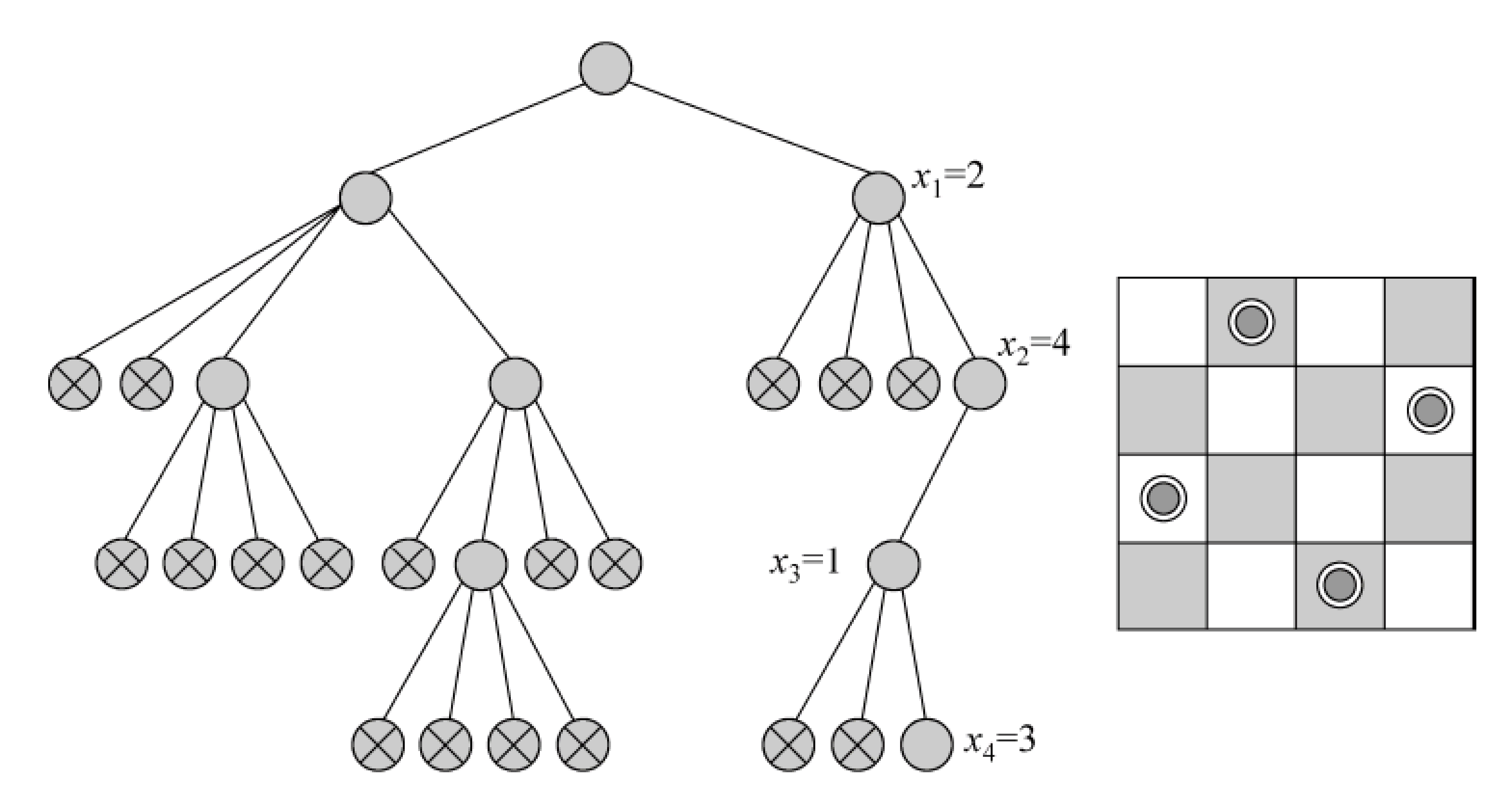


图 7-8　利用回溯解 7-皇后问题的一个例子

7.2.3　回溯算法的框架

1. 算法框架

虽然算法 7-4～算法 7-6 描述的是解决不同组合问题的回溯算法，观察可知，它们的基本框架是一致的。这是因为，首先，它们作为组合问题有相同的输入数据结构和输出数据结构。其次，它们的解空间被组织成搜索树结构。最后，对搜索树的搜索方式也都是按深度优先进行的。仔细考察，三者之间还是有些微妙的差别：首先，第 2 行对解向量的分量初始化值未必相同，算法 7-4 和算法 7-6 初始化为 0，而算法 7-5 却初始化为−1。当然算法 7-4、算法 7-6 第 12 行与算法 7-5 的第 15 行回溯前对 x_k 的恢复初始值也不尽相同。还有第 5 行的循环条件也不相同，算法 7-4 是 $k<m$，算法 7-5 是 $k<1$，而算法 7-6 是 $k<n$。这些差别是因为解向量的分量取值集合的差异引起的。其次，第 7 行判断部分解 $<x_1, x_2, \cdots, x_k>$ 是否合法的规则不同，算法 7-4 的是 $\forall 1\leqslant i\leqslant k((i, k)\in G[E] \rightarrow x_i\neq x_k)$，算法 7-5 的是 $\sum_{i=1}^{k} x_i a_i \leqslant C$，而算法 7-6 的是 $\forall 1\leqslant i<k(x_i\neq x_k \text{ and } |x_i-x_k|\neq|i-k|)$。第 8 行判断 $<x_1, x_2, \cdots, x_k>$ 是否为完整解的规则也不尽相同，算法 7-4 和算法 7-6 的是 $k=n$，算法 7-5 的是 $\sum_{i=1}^{k} x_i a_i = C$。这刚好反映了它们解决的是不同的组合问题。

这 3 个算法的异同，促使我们考虑能不能设计一个通用的解决组合问题的回溯算法？回答是肯定的。针对前面指出第一个区别点，只要将解向量的各分量取值集合作为参数传递给算法，就可以统一的方式逐一取得其中的每一个值。对于第二个区别点，可以将具体问判断部分合法解与判断合法完整解的规则写成特定的过程，而在回溯算法中加以调用。依据此思路，针对 7.1.2 节中所描述的一般组合问题，给出下列一般的回溯算法过程。

```
BACKTRACKITER(X1, X2, …, Xn)        ▷ Xk 是解向量 x 中分量 xk 的取值集合
1 solutions←∅
2 k←1                               ▷从第 1 层开始探索
3 while k≥1
```

```
4      do while X_k is not exhausted                         ▷探索第 k 层
5         do x_k ← next element in X_k
6            if IS-PARTIAL(x_1, x_2, …, x_k)                 ▷合法部分解
7               then if IS-COMPLETE(x_1, x_2, …, x_k)        ▷完整解,输出
8                       then solutions ← solutions ∪ {<x_1, x_2, …, x_k>}
9                            if X_k is not exhausted then continue this while loop
10                    if k = n                               ▷达到树叶,但非合法解
11                       then if X_k is exhausted
12                               then exit this while loop
13                       else k ← k+1                        ▷部分合法但不完整,进一步探索
14     reset X_k
15     k ← k-1                                               ▷回溯
16 return solutions
```

算法 7-7　回溯算法的框架

之所以把算法 7-7 称为回溯算法的框架,是因为要运行解决组合问题,需要向它"填充"以下东西。

(1) 解向量 x 的每一个分量的取值集合 $X_1, X_2, \cdots, X_n$ 作为传递给 BACKTRACKITER 的参数。

(2) 判断向量 $<x_1, x_2, \cdots, x_k>$ 是否为合法部分解的 IS-PARTIAL 过程,供 BACKTRACKITER 在第 6 行调用。

(3) 判断向量 $<x_1, x_2, \cdots, x_k>$ 是否为合法解的 IS-COMPLETE 过程,供 BACKTRACKITER 在第 7 行调用。

作为例子,我们对 3-着色问题给出刻画其特性的数据与过程。

为用算法 7-7 解决 3-着色问题,令集合 $X=\{1, 2, 3\}$,若图 $G=<V, E>$有 n 个顶点,则$\underbrace{X\times X\times\cdots\times X}_{n}$作为传递给 BACKTRACKITER 过程的参数。此外,还要编写下列两个过程。

```
IS-PARTIAL(x_1, x_2, …, x_k)
for i ← 1 to k-1
    do if (i, k) ∈ E[G] and x_i = x_k
          then return false
return true
```

```
IS-COMPLETE(x_1, x_2, …, x_k)
if k = n
    then return true
return false
```

算法 7-8　刻画 m-着色问题的 2 个特性过程

读者可仿照算法 7-8 写出子集和问题与 n-皇后问题的特性过程 IS-PARTIAL 和 IS-COMPLETE。

综上所述,用算法 7-7 解决组合问题,可以通过以下步骤进行。

(1) 将问题加以形式化描述,即确定作为输入数据的 n 个集合 $X_1, X_2, \cdots, X_n$,并将输

出数据描述为 n 维向量 $x=<x_1, x_2, \cdots, x_n>$，其中 $x_i \in X_i, i=1, 2, \cdots, n$。

(2) 根据问题特性，描述判断部分解、完整解的过程 IS-PARTIAL 和 IS-COMPLETE。

(3) 传递参数 $X_1, X_2, \cdots, X_n$，运行 BACKTACKITER 过程，调用 IS-PARTIAL 和 IS-COMPLETE 解决组合问题。

2. 程序实现

1) 数据的表示

对于组合问题的输入数据，根据前面的讨论可知，解空间 Ω 表示成解向量各分量取值集合 $X_i(i=1, 2, \cdots, n)$ 的笛卡儿积。在程序中，是一个一个分量逐一考察的，所以只要将每个分量的取值集合 X_i 表示出来就可以了。由于各分量的取值集合所含元素个数编程时并不知道，所以将它们表示成链表。对于每个分量的取值集合，算法中需要记忆各自当前的元素位置，以便于下一个元素的读取，所以还需要为每一个表示成链表的集合设置一个当前结点指针。下列定义的数据类型 ValueSet 就是用来表示解向量分量取值集合 X_i 的。

```
typedef struct{            /* 解向量分量取值集合类型 */
  LinkedList *list;        /* 存储一个分量所有可能取值的链表 */
  ListNode *current;       /* 当前访问的结点指针 */
}ValueSet;
```

程序 7-1　表示解向量取值集合的结构体类型

而将 Ω 直接表示成 ValueSet 类对象 $X_i(i=1, 2, \cdots, n)$ 的数组。由于 ValueSet 含有链表 LinkedList 类型的指针属性 list，所以需要有一个对不再使用的这类对象清理存储空间的函数。

```
1 void clrSet(ValueSet * set, void(*proc)(void *)){/* 清理分量取值集合的存储空间 */
2   clrList(set->list,proc);                      /* 清理链表属性的存储空间 */
3   free(set->list);
4 }
```

程序 7-2　清理解向量分量取值集合存储空间的 C 函数

程序中第 2 行调用第 2 章 2.1.3 节程序 2-3 中定义的函数 clrList 清理 ValueSet 对象 set 的 LinkedList * 属性 list，参数 proc 负责清理 list 中每个结点的 key 属性空间。

对输出数据，合法解向量用数组来表示是合适的。但合法解集合中元素个数事前不可知，故亦将其表示为链表。其次，考虑到即使在一个组合问题中合法解的长度未必相同，所以将合法解表示数组的同时，还需要表示出其长度。下列代码定义了表示合法解的数据类型。

```
typedef struct{              /* 合法解类型 */
  void *x;                   /* 解向量 */
  int k;                     /* 解向量长度 */
}Solution;
```

程序 7-3　表示合法解的结构体类型

由于 Solution 类型的数据对象含有 void * 类型属性，也需要有一个负责清理使用后存

储空间的函数。

```
void clrSolution(Solution * s){                      /* 清理解的存储空间 */
  if(s->x)
      free(s->x);
}
```

程序 7-4 负责清理 Solution 类型数据存储空间的函数

2) 回溯搜索函数

利用程序 7-1 和程序 7-3 定义的数据类型,可以把算法 7-7 的 BACKTRACKITER 过程实现为如下 C 函数。

```
LinkedList * backtrackiter(ValueSet * * OMG, int n){           /* 组合问题回溯求解 */
  int k=0, size=OMG[0]->list->eleSize;
  void * x=(void * )malloc(n * size);                          /* 解向量 */
  LinkedList * solutions=createList(sizeof(Solution),NULL);    /* 合法解集合 */
  assert(x&&solutions);
  while(k>=0){
      while(OMG[k]->current->next!=OMG[k]->list->nil){        /* X_k 未穷尽 */
          OMG[k]->current=OMG[k]->current->next;
          memcpy((char * )x+k * size,OMG[k]->current->key,size);
           if(isPartial(x, k)){                                /* 部分合法 */
               if(isComplete(x, k)){                           /* 完整合法解 */
                   Solution s;                                 /* 合法解暂存空间 */
                   s.x=(void * )malloc((k+1) * size);
                   memcpy(s.x,x,(k+1) * size);                 /* 保存解向量 */
                   s.k=k+1;                                    /* 解向量长度 */
                   listPushBack(solutions,&s);                 /* 加入解集 */
                   if(OMG[k]->current->next!=OMG[k]->list->nil)
                       continue;
               }
               if(k==n-1){                                     /* 达到树叶但非完整合法解 */
                   if(OMG[k]->current->next==OMG[k]->list->nil)
                       break;
               }else k++;
           }
      }
      OMG[k]->current=OMG[k]->list->nil;                      /* 重置 X_k */
      k--;                                                    /* 回溯 */
  }
  free(x);
  return solutions;                                           /* 返回合法解集合 */
}
```

程序 7-5 实现算法 7-7 的 C 函数

对程序 7-5 的说明如下。

(1) 函数 backtrackiter 有两个参数,第一个参数是 ValueSet * 类型数据的数组 OMG,表示解空间 Ω。第二个参数 n 表示数组 OMG 的元素个数。backtrackiter 返回表示为链表类型 LinkedList 的合法解集合。

(2) 第 2 行声明的 int 型变量 k,第 3 行声明的 **void** * 型变量 x,第 4 行声明的 LinkedList * 变量 solutions 的意义与算法 7-7 中的同名变量是一致的: k 表示探索层次,x 表示解向量,solutions 表示合法解集合。将 k 初始化为 0 是因为 C 语言中数组下标是从 0 开始的。为 x 分配空间时用来表示元素存储宽度的 size 值初始化为 OMG[0](表示 X_0)的链表中元素存储宽度。solutions 初始化为调用函数 createList 创建的空链表。函数 createList 定义于第 2 章 2.1.3 节的程序 2-3 中。

(3) 第 6～28 行的 **while** 循环对应算法 7-7 中第 3～14 行的 **while** 循环,循环条件变为 k≥0 是因为前面说过的原因:将就 C 语言中数组下标的编码规则。第 7～25 行的内嵌 **while** 循环对应算法 7-7 中第 4～12 行的内嵌 **while** 循环。该循环的循环条件利用 OMG[k] 的 current 属性是否指向表示 X_k 的链表的最后一个元素来表示 X_k 是否已经检测完。循环体中,第 8 行和第 9 行完成算法中第 5 行的操作 x_k←next element in X_k。第 10～24 行的 **if** 结构对应于算法中第 6～13 行的 **if** 结构。其中第 10 行和第 11 行调用的判断部分合法解函数 isPartial 及判断完整合法解函数 isComplete 需要事先声明其原型,对具体的组合问题编写合适的函数定义。第 12～16 行完成算法中第 8 行的操作 $solutions \leftarrow solutions \cup \{<x_1, x_2, \cdots, x_k>\}$。第 20～23 行的 **if-else** 结构对应算法中第 10～13 行的 **if-else** 结构。其中表达式 OMG[k]->current->next==OMG[k]->list->nil 表示条件 X_k is exhausted。

为便于重用,将程序 7-1、程序 7-3 对数据类型 ValueSet 和 Solution 的定义以及程序 7-2、程序 7-4 和程序 7-5 中各函数原型的声明写入文件夹 btrack 中的头文件 combineproblem.h 中,而将程序 7-2、程序 7-4 和程序 7-5 中各函数的定义写入同一文件夹中的源文件 backtrackiter.c 中。

3) 解 *m*-着色问题

有了函数 backtrackiter,要解决 m-着色问题只需要定义刻画该问题输入数据的解空间 Ω 和判断部分合法解函数 isPartial,以及判断合法完整解的函数 isComplete,然后调用 backtrackiter 就行了。

对于一个图 $G=<V, E>$, $V=\{1, 2, \cdots, n\}$, $E\subseteq V\times V$,可以用一个矩阵 $(a_{ij})_{n\times n}$ 来表示。其中 $a_{ij}=\begin{cases}1 & (i,j)\in E[G]\\ 0 & (i,j)\notin E[G]\end{cases}$, $1\leqslant i,j\leqslant n$,此矩阵称为图 G 的邻接矩阵。例如,图 7-6(a) 中的图 G 的邻接矩阵为

$$\begin{pmatrix}0 & 1 & 1 & 0 & 0\\ 1 & 0 & 0 & 1 & 1\\ 1 & 0 & 0 & 1 & 1\\ 0 & 1 & 1 & 0 & 1\\ 0 & 1 & 1 & 1 & 0\end{pmatrix}$$

把描述图 G 的邻接矩阵表示为一个按行优先存储(第 1 行元素存储完,存储第 2 行元素,……)的一维数组 G 中,图 G 的顶点个数表为 n,颜色数表为 m。为使各函数的参数表示

简洁,这 3 个数据对象均定义为全局量。例如,为表示图 7-6(a)的图 G 的 3-着色问题,有如下的全局变量 G、n、m 的定义。

```
int G[]={0,1,1,0,0,
         1,0,0,1,1,
         1,0,0,1,1,
         0,1,1,0,1,
         0,1,1,1,0},/*图 G 的邻接矩阵*/
   n=5,             /*图 G 的顶点个数*/
   m=3;             /*颜色种数*/
```

程序 7-6 表示图 7-6(a)的图 G 的 3-着色问题的数据定义

定义了颜色数 m 后,可以用如下的函数来构造解向量分量取值集合 X_i 和解空间 Ω。

```
1 ValueSet *newSet(){         /*创建集合 X*/
2   int i;
3   ValueSet *s=(ValueSet*)malloc(sizeof(ValueSet));
4   assert(s!=NULL);
5   s->list=createList(sizeof(int), NULL);
6   for(i=1;i<=m;i++)
7       listPushBack(s->list,&i);
8   s->current=s->list->nil;
9   return s;
10 }
11 ValueSet** makeOMG(){/*创建解空间 Ω*/
12   int i;
13   ValueSet **OMG=(ValueSet**)malloc(n*sizeof(ValueSet*));
14   assert(OMG!=NULL);
15   for(i=0;i<n;i++)
16       OMG[i]=newSet();
17   return OMG;
18 }
```

程序 7-7 用来创建 m-着色问题输入数据的 C 函数

对程序 7-7 的说明如下。

(1) 第 1～10 行定义的函数 newSet 针对全局变量 m 提供的颜色数量创建 m-着色问题中解向量分量取值的集合。第 3 行声明 ValueSet * 型变量 s 并为其分配存储空间。第 5 行调用函数 createList 为 s 的 list 属性创建一个空链表。第 6 行和第 7 行的 **for** 语句将{1, 2, …, m}添加到 list 中。第 8 行将 s 的 current 属性初始化为 list 的 nil 结点。由于 LinkedList 是双向链表,哑结点 nil 的 next 指针恰指向 list 的首结点。

(2) 第 11～18 行定义的函数 makeOMG 针对全局变量 n 提供的图 G 的顶点个数调用 n 次 newSet 创建 m-着色问题的解空间。第 13 行声明 ValueSet ** 型变量 OMG,并为其分配可存储 n 个 ValueSet * 型数据的空间而形成一个具有 n 个 ValueSet * 型元素的数组。第 15 行和第 16 行的 **for** 循环调用 newSet 函数为数组 OMG 的每个元素创建一个分量取值

集合。

下面来编写实现算法 7-8 中描述的刻画 m-着色问题的两个判断过程。

```
1 int isPartial(int *x, int k){
2   int i;
3   for(i=0;i<k;i++)
4       if(G[i*n+k]&&x[i]==x[k])   /*(i, k)∈E[G] and x_i=x_k */
5           return 0;
6   return 1;
7 }
8 int isComplete(int *x, int k){
9   return k==n-1;
10 }
```

程序 7-8　用于 m-着色问题的部分合法解判断函数及完整合法解判断函数

对程序 7-8 的说明如下。

(1) 第 1～7 行定义的函数 isPartial 实现的是算法 7-8 中的 IS-PARTIAL 过程。程序代码的结构与伪代码的十分相近。需要注意的是，伪代码中数组下标从 1 开始编码，而在 C 语言中，数组的下标是从 0 开始编码的，这就使得对应的循环条件边界有些微妙的差异。此外，我们是用一维数组按行优先表示矩阵，所以为访问矩阵的第 i 行第 j 列元素 G[i, j]，等价于访问数组中元素 G[i*n+j]。

(2) 第 8～10 行定义的函数 isComplete 实现的是算法 7-8 中的 IS-COMPLETE 过程。利用 C 语言中关系表达式的特点，程序代码比伪代码更简洁。

程序 7-6～程序 7-8 的代码写在文件夹 btrack 中的源文件 mcolor. c 中，读者可打开此文件研读。

7.3　子集树和排列树

在 7.2.3 节中，把 m-着色问题、n-皇后问题、子集和问题纳入到同一个算法的框架内。它使得人们能集中精力研究问题本身的特征，从而提高程序设计的效率。然而，算法框架的通用性是以算法的时空效率作为代价的：由于需要存储解向量的 n 个分量取值集合，当问题规模较大时，这是一个很大的空间开销。事实上，很多组合问题都有着一些特性，可以利用这些特性来提高算法的时空效率。本节就两种最常见的组合问题讨论它们的算法改进。

7.3.1　子集树问题

1. 问题的理解与描述

考虑子集和问题。其解是给定集合的子集，解向量 $<x_1, x_2, \cdots, x_k>(k \leqslant n)$ 是一个由 0 或 1 构成的比特串，其解空间可组织成一棵完全二叉树。这时，称这棵搜索树为一棵**子集树**，因为在这棵树中，从树根到任一片叶子的路径决定了一个解向量，而该向量决定了集合

的一个子集：$x_i=1$ 意味着子集中选取了第 i 个元素；$x_i=0$，意味着子集中未选取第 i 个元素。对搜索树为一棵完全二叉树的组合问题我们称其为子集树问题。形式化表示为如下。

输入：含有 n 个元素的集合 A。

输出：如果问题有合法解，输出所有合法解。表示这些解的向量 $x=<x_1, x_2, \cdots, x_k>$ 的 $k(\leqslant n)$ 个分量 $x_i\in\{0,1\}(i=1, 2, \cdots, k)$ 决定了 A 的一个子集。

子集树问题是一种常见的组合问题，除了前面介绍的子集和问题，下列**相容活动问题**也是一个典型的子集树问题。

n 个活动 $\{a_1, a_2, \cdots, a_n\}$ 竞争一个资源。每个活动 a_i 有开始时间 s_i 和完成时间 f_i。任何时刻，资源只能被一个活动所占用。两个活动 a_i 和 a_j，活动时间区间 $[s_i, f_i)$ 和 $[s_j, f_j)$ 满足 $[s_i, f_i)\cap[s_j, f_j)=\varnothing$，称活动 a_i 和 a_j 相容。相容活动问题描述如下。

输入：n 个活动 $A=\{a_1, a_2, \cdots, a_n\}$ 的开始时间 $s=\{s_1, s_2, \cdots, s_n\}$ 和完成时间 $f=\{f_1, f_2, \cdots, f_n\}$。不失一般性，设 $f_1\leqslant f_2\leqslant\cdots\leqslant f_n$。

输出：两两相容的活动组成的集合。

例如，11 个活动如图 7-9 所示。s_i 表示活动 a_i 的开始时间，f_i 表示 a_i 的完成时间。则 $\varnothing$、$\{a_i\}(1\leqslant i\leqslant n)$、$\{a_1, a_4\}$、$\{a_4, a_8, a_{11}\}$、$\{a_1, a_4, a_8, a_{11}\}$ 都是问题的合法解。A 的这些子集都可以表示为比特串。例如，$\{a_4, a_8, a_{11}\}$ 可表为 $<0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1>$。因此，活动相容问题可视为一个子集树问题。

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

图 7-9　一组竞争同一资源活动的时间表

2. 算法的伪代码描述

1) 子集树搜索算法

对于子集树问题，由于搜索树为一棵完全二叉树，可以将搜索过程加以简化。

```
SUBSET-TREE(n)
solutions←∅
for each 1≤i≤n set x_i as −1
k←1                                                   ▷从第 1 层开始探索
while k≥1
    do while x_k<1
        do x_k←x_k+1                                  ▷x_k∈X={0,1}
          if IS-PARTIAL (x_1, x_2, …, x_k)            ▷部分合法
            then if IS-COMPLETE (x_1, x_2, …, x_k)    ▷合法解
                   then solutions←soulutions∪{<x_1, x_2, …, x_k>}
                        if x_k<1 then continue this while loop
                 else if k=n                          ▷达到树叶，但并非合法
                        then if x_k≥1
                               then exit this while loop
```

```
14                    else k←k+1              ▷合法但不完整，进一步探索
15     x_k←−1
16     k←k−1                                  ▷回溯
17 return solutions
```

算法 7-9　在子集树中回溯探索解向量的过程

与算法 7-7 相比，子集树问题的解向量一定是整型数组。由于解向量分量取值集合为确定的 $X=\{0, 1\}$，故无须将 n 个 X 作为算法过程的参数。这样就提高了空间效率。事实上，算法 7-9 更接近于解决子集和问题的算法 7-5。除了第 7 行和第 8 行两行对于部分合法解的判断和完整合法解的判断不同外，其余部分完全一致。这两个不同点恰好说明了算法 7-5 是解决子集和问题的算法，而算法 7-9 是用来解决所有子集树问题的通用算法。只要对具体的子集树问题写出判断部分合法解的过程 IS-PARTIAL 和判断完整合法解的过程 IS-COMPLETE，就可以调用算法 7-9 解决该问题。下面以相容活动问题为例，说明这一解题过程。

2）相容活动合法解判断算法

根据对相容活动问题的理解，将该问题的解表示为分量取值于$\{0, 1\}$的向量，第 i 个分量 $x_i=1$ 则意味着活动 a_i 被选择，$x_i=0$ 表示活动 a_i 未被选择。所谓活动 a_i 与 a_j 相容，指的是活动时间区间$[s_i, f_i)\cap[s_j, f_j)=\varnothing$。由于假定活动按完成时间的升序排列，所以 a_i 与 a_j 相容条件可等价地表为$(i<j)\to(f_i\leqslant s_j)$。于是，对于向量$<x_1, x_2, \cdots, x_k>$在已知$<x_1, x_2, \cdots, x_{k-1}>$部分合法的前提下，检测其是否部分合法，只需找到$<x_1, x_2, \cdots, x_{k-1}>$中最后一个不等于 0 的分量 x_i，计算是否 $f_i\leqslant s_k$。而在$<x_1, x_2, \cdots, x_k>$部分合法的前提下，检测其是否完整合法，只需检测是否 $k=n$。据此，对活动相容问题可以写出以下过程。

```
IS-PARTIAL(x_1, x_2, …, x_k)
if x_k=0          ▷<x_1, x_2, …, x_{k-1}>部分合法必有<x_1, x_2, …, x_{k-1}, 0>部分合法
    then return true
i←k−1
while x_i=0       ▷找到<x_1, x_2, …, x_{k-1}>中最后一个不等于 0 的分量
    do i←i−1<x_1, x_2, …, x_{k-1}>
if i≤0            ▷<0, 0, …, 0, x_k>必部分合法
    then return true
if f_i≤s_k        ▷a_i 与 a_k 相容
   return true
return false
IS-COMPLETE(x_1, x_2, …, x_k)
if k=n            ▷到达叶子
    then return true
return false
```

算法 7-10　判断相容活动部分合法解及完整合法解的算法过程

很容易看出，过程 IS-PARTIAL 的运行时间是 $O(n)$，而过程 IS-COMPLETE 的运行时间是 $O(1)$。

3. 程序实现

1) 子集树回溯搜索

利用程序 7-3 定义的合法解类型 Solution，可以把算法 7-9 实现为如下函数。

```
LinkedList * subSetTree(int n){
  LinkedList *solutions=createList(n*sizeof(Solution),NULL);/*合法解集合*/
  int *x=(int*)malloc(n*sizeof(int)),k=0;
  assert(x&&solutions);
  memset(x,-1,n*sizeof(int));
  while(k>=0){
      while(x[k]<1){                              /*Xk 未穷尽*/
          x[k]++;
          if(isPartial(x, k)){                    /*部分合法*/
              if(isComplete(x, k)){               /*完整合法解*/
                  Solution s;                     /*合法解暂存空间*/
                  s.x=(int*)malloc((k+1)*sizeof(int));
                  memcpy(s.x,x,(k+1)*sizeof(int)); /*保存解向量*/
                  s.k=k+1;                        /*解向量长度*/
                  listPushBack(solutions,&s);     /*加入解集*/
                  if(x[k]<1)                      /*本层尚未检测完*/
                      continue;
              }
              if(k>=n-1){                         /*达到树叶但非合法解*/
                  if(x[k]==1)
                      break;
              }else k++;
          }
      }
      x[k]=-1;                                    /*重置 Xk*/
      k--;
  }
  free(x);
  return solutions;                               /*返回合法解集合*/
}
```

程序 7-9 实现算法 7-9 的 SUBSET-TREE 过程的 C 函数

对程序 7-9 的说明如下。

(1) 由于仅对子集树进行搜索，故无须从参数传递搜索树的信息。函数 subSetTree 仅含有一个表示解向量长度的参数n。与 backtrackiter 一样，subSetTree 返回表示合法解集合的链表。

(2) 函数中声明的变量 solutions、k、x 的意义与算法 7-9 中的同名变量一致。注意，第

5 行调用库函数 memset① 将 x 的所有分量初始化为−1。

(3) 与程序 7-5 相比，仅有对 X_k 是否穷尽的检测条件(见第 7 行、第 16 行和第 20 行)和对解向量分量赋值方式(见第 8 行、第 25 行)稍有改变，其余都是一样的。

为便于代码重用，将函数 subSetTree 的原型声明加入到文件夹 btrack 中的头文件 combineproblem.h 中，并将该函数的定义写入同一文件夹中的源文件 subsettree.c 中。

2）解相容活动问题

下面用程序 7-9 来解决相容活动问题。首先，需要定义描述问题各个活动开始时间和完成时间的数组 s、f 以及活动个数 n。例如，表示图 7-9 中各个活动时间的 s、f 和 n 的声明如下。为使各功能函数的参数简洁，将这些变量定义为全局变量。

```
int s[]={1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12},
    f[]={4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14},
    n=11;
```

有了这些数据的定义，可以把算法 7-10 中判断相容活动合法部分解与合法完整解过程实现为下列函数。

```
int isPartial(int *x, int k){
   int i=k-1;
   if(x[k]==0)
       return 1;
   while(i>=0&&x[i]==0)
       i--;
   if(i<0)
       return 1;
   return f[i]<=s[k];
}
```

程序 7-10　实现算法 7-10 中判断相容活动合法部分解与合法完整解过程的函数

把上述的全局变量 s、f、n 的定义，函数 isPartial、isComplete 的定义以及调用 subSetTree 解决图 7-9 中表示的相容活动问题的程序存储在文件夹 btrack 中的源文件 compatableactives.c 中，读者可打开文件研读并试运行。

7.3.2　排列树问题

1. 问题理解与描述

仔细考察 n-皇后问题，解向量的第 k 个分量只能选取前 $k-1$ 个分量尚未取到过的列号值。换句话说，一个完整的解向量恰是 1, 2, …, n 的一个排列。然而，在 7.3.1 节中，将每个分量的取值集合设置为{1, 2, …, n}。因此，搜索树成为一棵完全 n 叉树(见图 7-5)。事实上，可以把 n-皇后问题的搜索树从 n 叉完全树简化为根结点有 n 个孩子，这些孩子结点

① 库函数 memset 的原型声明在头文件 string.h 中。

每一个有 $n-1$ 个孩子,最后一层,每个结点只有 1 个孩子。例如,4-皇后问题简化后的搜索树如图 7-10 所示。

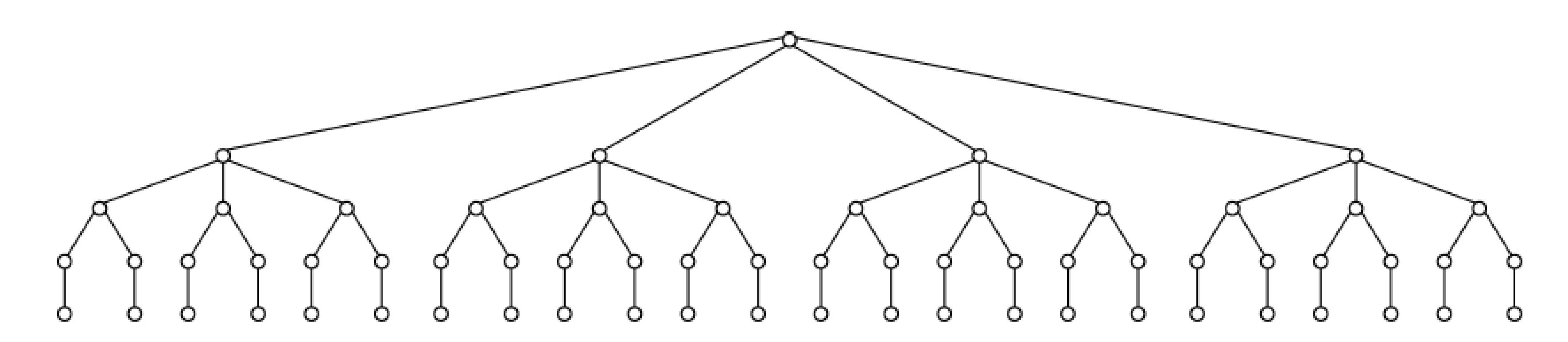


图 7-10　4-皇后问题简化后的搜索树

把解向量是 n 个元素的排列的组合问题的搜索树称为排列树,而把这一类组合问题称为**排列树问题**。形式化表示为如下。

输入:n 个各不相同的元素构成的集合 $A=\{a_1, a_2, \cdots, a_n\}$。

输出:所有 A 的 n 个元素的排列 $<x_1, x_2, \cdots, x_n>$,使得 $f(x_1, x_2, \cdots, x_n)=$true。其中,$f$ 是定义在 Ω 为所有 A 的 n 个元素的排列组成的集合,值域是{true, false}的函数。

n-皇后问题是典型的排列树问题。由于排列树中的结点数为 $\Theta(n!)$①,若检测合法解的过程耗时为 $\Theta(n)$,则排列树问题的回溯算法耗时 $\Theta(nn!)$,这当然要比搜索树为 n 叉完全树问题的回溯算法耗时 $\Theta(n^{n+1})$好得多。

排列树问题是一类常见的组合问题,例如,下列著名的 Hamilton 回路问题也可以归纳成排列树问题。

Hamilton 回路问题:设无向图 $G=<V, E>$。其中,$V=\{1, 2, \cdots, n\}$,$E\subseteq V\times V$。寻求从顶点 $s\in V$ 出发,经过 G 中每一个顶点一次,最后回到顶点 s 的回路,如图 7-11(a)所示,其中用带阴影的边表示的就是该图中这样的一条路径。这个问题称为 Hamilton 回路问题。不是每个无向图都存在 Hamilton 回路,例如,图 7-11(b)中就不存在 Hamilton 回路。

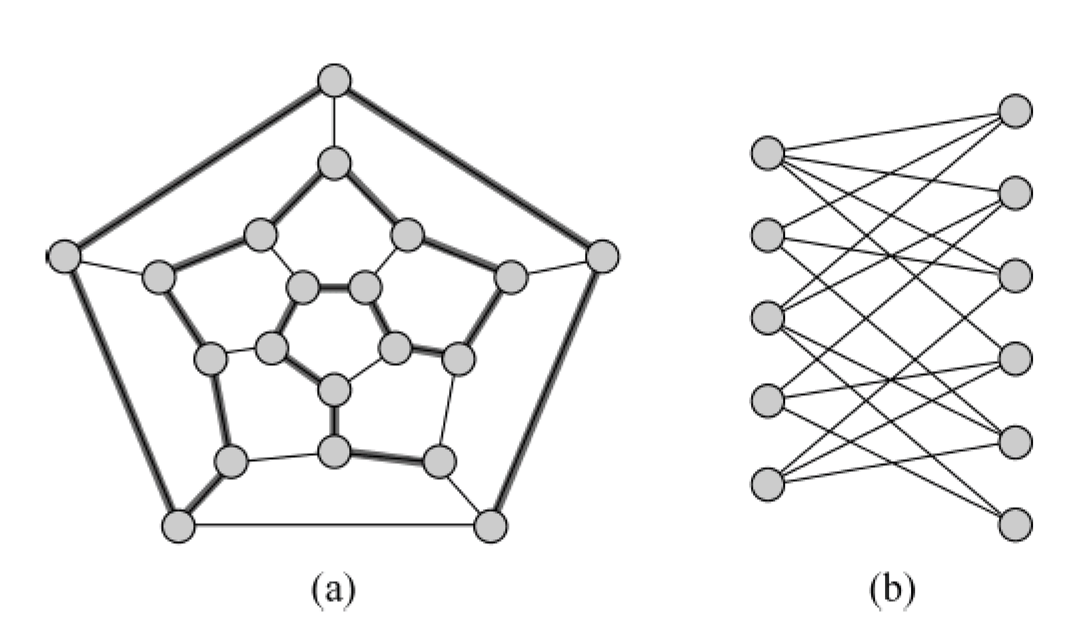


图 7-11　图的 Hamilton 回路

Hamilton 回路问题可形式化地描述为如下。

输入:具有 n 个顶点的无向图 $G=<V, E>$,起始顶点 s。

输出:如果 G 具有 Hamilton 回路,输出所有向量 $x=<x_1, x_2, \cdots, x_n>$,其中,$x_1=s$,$x_2, \cdots, x_n$ 是 $1, 2, \cdots, n$ 中除了 s 以外的 $n-1$ 个数的一个排列,且 $x_1, x_2, \cdots, x_n, x_1$ 是

① 根据 Stirling 公式知 $n!$ 也是指数级的量。

G 中的一条回路;否则,输出空集。

图 7-12(a)中给出了一个具有 5 个顶点无向图 G,图 7-12(b)展示了对 G 计算从顶点 $s=1$ 出发的 Hamilton 回路问题的搜索树。

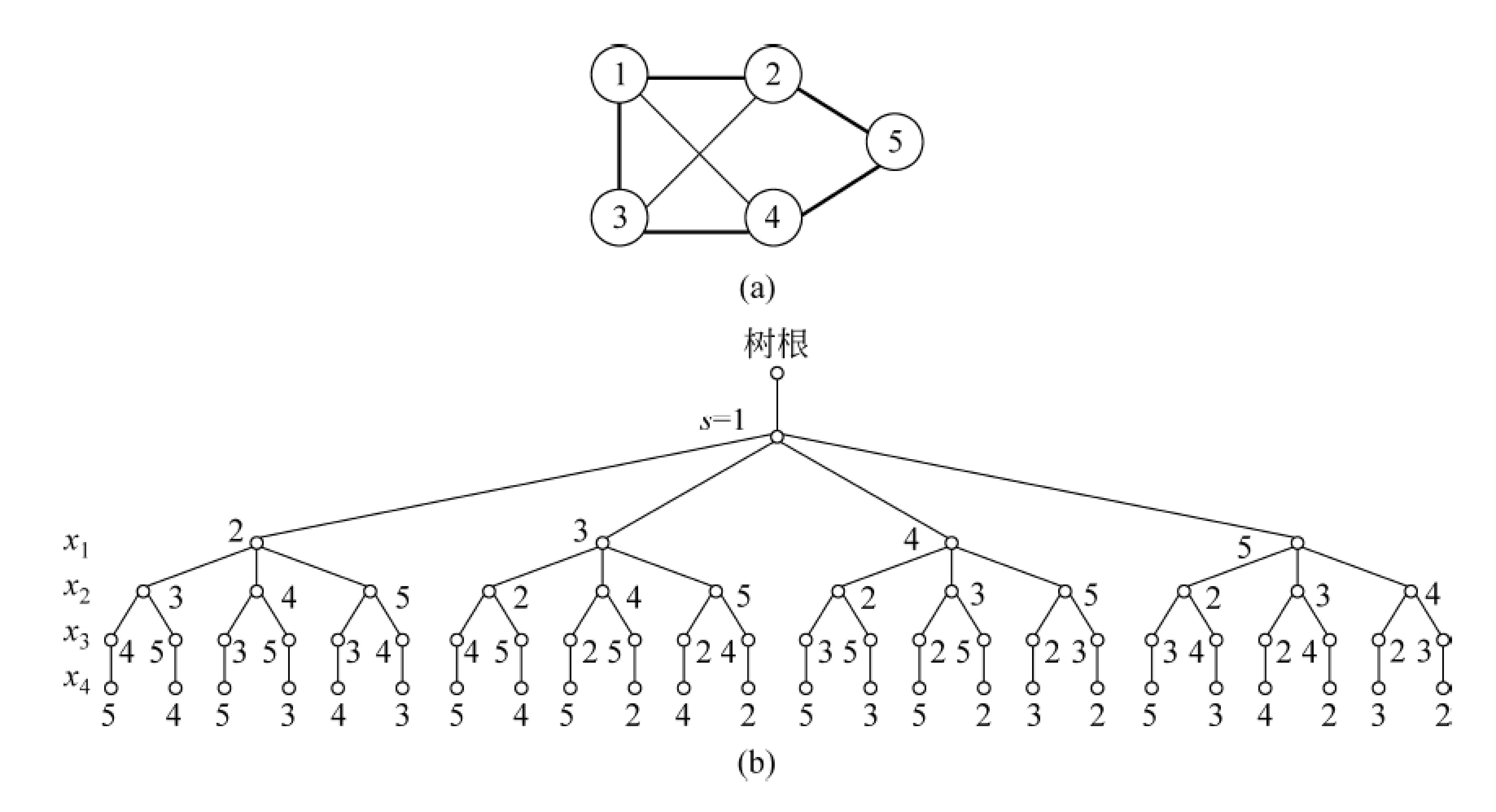


图 7-12　一个无向图的 Hamilton 回路问题的搜索树

2. 算法的伪代码描述

1）创建排列树

解决排列树问题的前提是将解空间 Ω 构建成排列树。下列的递归过程能生成一棵排列树。

```
PERMUTE (x, k)
1 if k>n
2     then solutions← solutions∪{<x1, x2, …, xn>}
3         return
4 for i←k to n                    ▷对当前第 k 个分量逐一取得各种可能的值
5     do xi↔xk                    ▷交换 xi 和 xk
6         PERMUTE(x, k+1)
7         xi↔xk                   ▷还原 xi 和 xk,准备创建下一个不同的排列
```

算法 7-11　创建 n 个元素全排列的递归算法

算法 PERMUTE 过程运行如下。当参数 $k>n$ 时,意味着 $<x_1, x_2, \cdots, x_n>$ 已构成一个排列,输出后回溯。对 $k\leqslant n$ 的情况,第 4～7 行的 **for** 循环对第 k 层的 $n-k+1$ 个值逐一取得,并递归到下一层。

由于算法 7-11 描述的产生 n 个元素全排列的递归算法并非末尾递归(第 6 行递归调用后还有第 7 行的交换元素的操作),所以要将其转换成等价的迭代版本需要借助于栈数据结构。

```
PERMUTE (origin)
1 S←∅, solutions←∅
```

```
2 copy origin to x
3 k←1, i←0
4 PUSH(S, i)
5 while k≥1
6     do i←i+1
7        while i≤n
8            do xi↔xk
9               if k≥n
10                 then solutions← solutions∪{<x1, x2, …, xk>}
11                      exit from this while loop
12              k←k+1
13              PUSH(S, i)
14              i←k
15      k←k-1
16      i←POP(S)
17      xi↔xk
18 return solutions
```

算法 7-12　创建 n 个元素全排列的迭代算法

与算法 7-11 相比,算法 7-12 含有一个参数 *origin*,表示 n 个元素的原始排列。该过程用两个嵌套的 **while** 循环模拟递归。其中,栈 S 扮演递归过程中系统栈暂存层次信息的角色:往下一层探索之前,第 13 行将本层当前取值位置信息 i 压栈;每次回溯前,将上层取值位置信息 i 出栈。

2) 排列树回溯搜索

利用产生 n 个元素全排列的算法框架 PERMUTE,不难写出下列对排列树回溯搜索的过程。

```
PERMUTE-TREE(origin)
S←∅, solutions←∅
copy origin to x
k←1, i←0
PUSH(S, i)
while k≥1
    do i←i+1
       while i≤n
           do xi↔xk
              if IS-PARTIAL(x1, x2, …, xk)
                then if IS-COMPLETE(x1, x2, …, xk)
                        then solutions← solutions∪{<x1, x2, …, xk>}
                             exit from this while loop
              k←k+1
              PUSH(S, i)
              i←k
    k←k-1
    i←POP(S)
```

```
    x_i ↔ x_k
return solutions
```

算法 7-13　对排列树回溯搜索过程

3）解 Hamilton 回路问题

首先回顾 Hamilton 回路问题的输入输出数据。输入数除了表示图 G 的邻接矩阵外，还有一个作为起点的整数 s。而作为输出的向量 $<x_1, x_2, \cdots, x_n>$ 中，$x_1=s$ 是固定的。所以，调用算法 PERMUTE-TREE 时，传递给它的参数 *origin* 是 1，2，…，n 中剔除 s 后的 $n-1$ 个元素。针对 Hamilton 回路问题的这些特点，写出如下判断部分合法解与完整合法解过程。

```
IS-PARTIAL(x, k)
if (s, x_1) ∉ E(G)                      ▷起点与顶点不构成 G 的一条边
    then return false
for i←1 to k-1                          ▷检验< x_1, …, x_k >的合法性
    if (x_i, x_{i+1}) ∉ E[G]            ▷(x_i, x_{i+1})不是 G 的边
        then return false
return true
IS-COMPLETE(x, k)
if k=n-1 and (x_{n-1}, s) ∈ E[G]
    then return true
return false
```

算法 7-14　刻画 Hamilton 回路问题特征的过程

算法 7-14 中的 IS-PARTIAL 过程运行时间是 $O(n)$，IS-COMPLETE 过程的运行时间是 $O(1)$。

3. 程序实现

1）排列树回溯搜索

和程序 7-9 相似，利用程序 7-3 定义的合法解类型 Solution，可将算法 7-13 的 PERMUTE-TREE 过程实现为如下函数。

```
LinkedList * permuteTree(void * origin, int size, int n){
  Stack * S=createStack(sizeof(int));
  void * x=(void * )malloc(n * size);
  int k=0,i=-1;
  LinkedList * solutions=createList(n * sizeof(Solution),NULL);/ * 合法解集合 * /
  assert(x&&solutions);
  memcpy(x,origin,n * size);
  k=0;push(S,&i);
  while(k>=0){
      i++;
      while(i<n){
         swap((char * )x+k * size,(char * )x+i * size,size);
```

```
13            if(isPartial(x, k))
14                if(isComplete(x,k)){
15                    Solution s;                              /* 合法解暂存空间 */
16                    s.x=(void *)malloc((k+1) * size);
17                    memcpy(s.x,x,(k+1) * size);              /* 保存解向量 */
18                    s.k=k+1;                                 /* 解向量长度 */
19                    listPushBack(solutions,&s);              /* 加入解集 */
20                    break;
21                }
22            k++;
23            push(S,&i);
24            i=k;
25        }
26        k--;
27        i= *((int *)(pop(S)->key));
28        swap((char *)x+k * size,(char *)x+i * size,size);
29    }
30    free(x);
31    clrStack(S,NULL);
32    free(S);
33    return solutions;                                        /* 返回合法解集合 */
34 }
```

程序 7-11　实现算法 7-13 的 C 函数

对程序 7-11 的说明如下。

(1) 和算法一样,函数 permuteTree 有一个表示 n 个元素原始排列的参数 origin。参数 size 表示 origin 中元素的存储宽度。函数返回表示合法解集合链表。

(2) 变量 S、x、solutions、i、k、n 的意义与算法过程中的同名变量一致。其中,栈 S 是第 2 章 2.2.2 节程序 2-11 定义的 Stack 类型。解向量 x 的类型定义成 **void** * 是因为我们希望该函数能处理不同类型数据的排列树问题。

(3) 程序代码的结构与算法伪代码结构十分接近。注意第 12 行和第 28 行调用第 3 章 3.1.2 节程序 3-3 定义的函数 swap,执行算法 7-13 中第 8 行和第 18 行的交换 x_i 和 x_k 操作 $x_i \leftrightarrow x_k$。

为便于代码重用,将函数 permuteTree 的原型声明写入 btack 文件夹的头文件 combineproblem.h,而将程序 7-11 存储到 btrack 文件夹中的源文件 permutetree.c 中。

2) 解 Hamilton 回路问题

要解决 Hamilton 回路问题,需要用来表示图 G 的邻接矩阵 $G[]$,图 G 的顶点个数 n、作为起点的顶点 s 以及 1,2,…,n 中剔除 s 后的 n 个整数的元素排列数组 origin。和解 m-着色问题时相似,将这些数据定义成全局变量,且将图 G 的邻接矩阵表示成按行优先存储的一维数组。

```
int G[]={0,1,1,1,0,
         1,0,1,0,1,
```

```
        1,1,0,1,0,
        1,0,1,0,1,
        0,1,0,1,0},                        /* 图 G 的邻接矩阵 */
    n=5,                                   /* 图 G 的顶点个数 */
    s=1,
    origin[]={2, 3, 4,5};
```

做了这些数据准备后，将算法 7-14 的两个过程实现为如下函数。

```
1 int isPartial(int *x, int k){
2   int i;
3   if(G[(s-1)*n+x[0]-1]==0)               /* (s, x1)∉E(G) */
4       return 0;
5   for(i=0;i<k;i++)
6       if( G[(x[i]-1)*n+x[i+1]-1]==0)     /* (xi, xi+1) ∉E[G] */
7           return 0;
8   return 1;
9 }
10 int isComplete(int *x, int k){
11   return (k==n-2)&&G[(x[n-2]-1)*n+s-1];   /* k=n-1 and (xn-1, s) ∈E[G] */
12 }
```

程序 7-12　实现算法 7-14 的 C 函数

对程序 7-12 的说明如下。

(1) 在 Hamilton 回路问题的数据描述中，n 是表示图 G 的顶点个数。而调用 permuteTree 时，传递给它的 origin 数组中仅含剔除 s 后的 n−1 个元素，要产生的排列树的层数为 n−1。所以，解向量 x 的长度为 n−1，即 x[0..n−2]。

(2) 第 1～9 行定义的函数 isPartial 实现的是算法 7-14 中的 IS-PARTIAL 过程。图 G 的邻接矩阵元素 G[s, x_1]的值表示 (s, x_1)是否为图 G 的边。由于 C 语言中数组下标是从 0 开始的，所以 G[s, x_1]表为 G[s−1][x[0]−1]，由于是用一维数组表示矩阵，所以 G[s−1][x[0]−1]表为 G[(x[i]−1)*n+x[i+1]−1]。同样，G[(x[i]−1) *n+x[i+1]−1]表示(x_i, x_{i+1})是否为图 G 的边。

(3) 第 10～12 行定义的函数 isComplete 实现算法 7-14 中的 IS-COMPLETE 过程。根据(1)中对数据定义的说明，k==n−2 表明 x[0..k]已经构成搜索树中一条到达树叶的路径。类似地说明(2)，G[(x[n−2]−1)*n+s−1]不等于 0 表示(x_k, s)是图 G 的一条边。

程序 7-12 及调用程序 7-11 解决 Hamilton 回路问题的代码存储在 btrack 文件夹的源文件 hamiltoncircuit.c 中，读者可打开研读并试运行。

7.3.3　应用

本节讨论两个可以用回溯算法解决的组合问题。

1. Queens in Peaceful Positions

Description

On a chessboard of size $N \times N$ ($N \leqslant 50$) N queens are placed. We'll say that queens are in peaceful position if none of them can attack another. You are to find the total amount of peaceful positions that can be obtained from the given peaceful position by rearranging of exactly three queens.

Input

The first line of input will contain an integer number N that represents the size of a chessboard (and the number of queens also). It will be followed by N lines describing positions of queens. Each line will contain two integers X and Y separated by a space. These numbers represent horizontal and vertical coordinates and lay in a range from 1 to N.

Output

The output consists of a single integer representing the number of peaceful positions that can be achieved from initial position by moving of exactly three queens. Note: queens are not numbered so if you rearrange them on the chessboard using only squares they already occupied you'll always get the same peaceful position, not the new one.

Sample Input

```
4
2 1
1 3
3 4
4 2
```

Sample Output

```
0
```

1）问题的理解与算法描述

对给定规模为 $N \times N$ 的棋盘上的一个 N-皇后问题格局 $<x_1, x_2, \cdots, x_N>$，要求找出该格局恰移动 3 个皇后的位置可得到 N-皇后问题可行解的个数。解决此问题的一个思路是计算出 N-皇后问题的所有可行解，然后逐一与 x 比较，计算与 x 对应行不同的位置数，统计不同位置数为 3 的可行解个数。

```
QUEENS-IN-PEACEFUL-POSITION(configuration)
origin←<1, 2, …, N>
solutions←PERMUTE-TREE(origin)
count←0
for each p ∈ soulutions
    do if COMPARE(configuration, p)=3
        then count← count+1
```

```
return count
```

算法 7-15　移动 3 个皇后的位置后问题可行解算法

其中调用算法 7-13 的搜索过程 PERMUTE-TREE 时需要用到如下判断 N-皇后问题的部分可行解与完整可行解的过程。

```
IS-PARTIAL(x1, x2, …, xk)
for i←1 to k-1
    do for j←i+1 to k
        do if |xi-xj|=|i-j|
            then return false
return true
IS-COMPLETE(x1, x2, …, xk)
if k<N
    then return false
return true
```

算法 7-16　判断 N-皇后问题的部分可行解与完整可行解的算法

此外，QUEENS-IN-PEACEFUL-POSITION 过程调用的计算两个格局的不同位置皇后数的 COMPARE 过程伪代码如下。

```
COMPARE(x, y)
diffrent←0
for i←1 to N
    do if xi≠yi
        then diffrent← diffrent+1
return different
```

算法 7-17　COMPARE 的伪代码

2）**程序实现**

将上述各算法实现为如下 C 程序。

```
int *origin,                          /*棋盘原始格局*/
  n;                                  /*棋盘规模*/
int isPartial(int *x, int k){
  int i,j;
  for(i=0;i<k;i++)
     for(j=i+1;j<=k;j++)
        if((x[i]-x[j]==j-i) || (x[j]-x[i]==j-i))
           return 0;
  return 1;
}
int isComplete(int *x, int k){
  return k==n-1;
}
int compare(int *x, int *y){
```

```
15    int i,different=0;
16    for(i=0;i<n;i++)
17        if(x[i]!=y[i])
18            different++;
19    return different;
20 }
21 int main(){
22    int i, * configuration, x, y, count=0;
23    LinkedList * solutions;
24    ListNode * p;
25    FILE * f1=fopen("chap07/Queens in peaceful positions/inputdata.txt","r"),
26         * f2=fopen("chap07/Queens in peaceful positions/outputdata.txt","w");
27    assert(f1&&f2);
28    fscanf(f1,"%d",&n);
29    assert(origin=(int * )malloc(n * sizeof(int)));
30    assert(configuration=(int * )malloc(n * sizeof(int)));
31    for(i=0;i<n;i++){
32        origin[i]=i+1;
33        fscanf(f1, "%d %d",&x, &y);
34        configuration[y-1]=x;
35    }
36    solutions =permuteTree(origin, sizeof(int), n);
37    for(p= solutions ->nil->next;p!= solutions ->nil; p=p->next){
38        if(compare(configuration,(int * )((Solution * )(p->key))->x)==3)
39            count++;
40    }
41    fprintf(f2, "%d\n",count);
42    free(configuration);free(origin);
43    clrList(solutions, clrSolution);
44    free(solutions);
45    fclose(f1);fclose(f2);
46    return 0;
47 }
```

程序 7-13　实现后的程序

对程序 7-13 的说明如下。

(1) 第 1 行和第 2 行将表示棋盘原始格局与棋盘规模的变量 origin 和 n 定义成全局变量,可以使各功能函数的参数比较简洁。

(2) 第 3～10 行定义的函数 isPartial,第 11～13 行定义的函数 isComplete 和第 14～20 行定义的函数 compare 实现的是上述的同名算法过程。代码结果几乎一致,读者可比较研读。

(3) 第 21～47 行定义的 main 函数,实现算法过程 QUEENS-IN-PEACEFUL-POSITION,解决 Queens in Peaceful Positions 问题。函数题中定义的变量 configuration、solution、count、p 与算法中的同名变量意义一致。文件指针 f1、f2 分别指向输入、输出文件。第 31～35 行的 **for** 循环从 f1 中读取数据初始化 conficuration,同时初始化 origin。第 36 行调用函

数 permuteTree，用回溯方法解 n-皇后问题。第 37～40 行的 **for** 循环对 solutions 中的每个解 p 调用函数 compare 计算其与 configuration 的差别，并用 count 计数两者相差 3 个皇后位置的情形。第 41 行将 count 写入 f2 文件。

上述程序存储在文件夹 chap07/Queens in peaceful positions 中的源文件 Queensinpeacefulpositions. c 中。读者可打开文件研读并试运行。

2. Color the Map

You were lucky enough to get a map just before entering the legendary magical mystery world. The map shows the whole area of your planned exploration, including several countries with complicated borders. The map is clearly drawn, but in sepia ink only; it is hard to recognize at a glance which region belongs to which country, and this might bring you into severe danger. You have decided to color the map before entering the area. "A good deal depends on preparation," you talked to yourself.

Each country has one or more territories, each of which has a polygonal shape. Territories belonging to one country may or may not "touch" each other, i. e. there may be disconnected territories. All the territories belonging to the same country must be assigned the same color. You can assign the same color to more than one country, but, to avoid confusion, two countries "adjacent" to each other should be assigned di. erent colors. Two countries are considered to be "adjacent" if any of their territories share a border of non-zero length.

Write a program that finds the least number of colors required to color the map.

Input

The input consists of multiple map data. Each map data starts with a line containing the total number of territories n, followed by the data for those territories. n is a positive integer not more than 100. The data for a territory with m vertices has the following format:

String

x_1 y_1

x_2 y_2

⋮

x_m y_m

−1

"String" (a sequence of alphanumerical characters) gives the name of the country it belongs to. A country name has at least one character and never has more than twenty. When a country has multiple territories, its name appears in each of them.

Remaining lines represent the vertices of the territory. A vertex data line has a pair of nonnegative integers which represent the x-and y-coordinates of a vertex. x-and y-coordinates are separated by a single space, and y-coordinate is immediately followed by a newline. Edges of the territory are obtained by connecting vertices given in two adjacent

vertex data lines, and by connecting vertices given in the last and the first vertex data lines. None of x-and y-coordinates exceeds 1000. Finally, −1 in a line marks the end of vertex data lines. The number of vertices m does not exceed 100.

You may assume that the contours of polygons are simple, i.e. they do not cross nor touch themselves. No two polygons share a region of non-zero area. The number of countries in a map does not exceed 10.

The last map data is followed by a line containing only a zero, marking the end of the input data.

Output

For each map data, output one line containing the least possible number of colors required to color the map satisfying the specified conditions.

Sample Input

```
6
Blizid
0 0
60 0
60 60
0 60
0 50
50 50
50 10
0 10
-1
Blizid
0 10
10 10
10 50
0 50
-1
Windom
10 10
50 10
40 20
20 20
20 40
10 50
-1
Accent
50 10
50 50
35 50
35 25
-1
```

```
Pilot
35 25
35 50
10 50
－1
Blizid
20 20
40 20
20 40
－1
4
A1234567890123456789
0 0
0 100
100 100
100 0
－1
B1234567890123456789
100 100
100 200
200 200
200 100
－1
C1234567890123456789
0 100
100 100
100 200
0 200
－1
D123456789012345678
100 0
100 100
200 100
200 0
－1
0
```

Sample Output

```
4
2
```

1）问题理解

一个国家由若干个区域组成，每个区域的边界由若干条直线段构成一个多边形，边界中的线段的端点在输入文件中表示成横坐标及纵坐标（整数）。利用这些数据，可以绘制成一幅探宝图。为更清晰地考察这份探宝图，对地图着色。同一国家的所有区域着同一种颜色。

相邻的不同国家不能着同一颜色。按这样的要求,对 n 个区域构成的若干个国家的地图进行最少颜色的着色。

解本题的想法之一是将地图中各区域构成的 n 个国家间相邻关系表示成一个图 G,图 7-13(a)为由 5 个区域构成的 4 个城市的地图。图 7-13(b)表示图 7-13(a)中地图的 4 个城市相邻关系的图。然后利用 m-着色问题的回溯探索算法(见 7.2.3 节的算法 7-7 和算法 7-8),按下列方式计算出最少用色 m。

```
m←0
repeat
  m←m+1
  solution← BACKTRACKITER(G, n, m)
until solutions≠∅
```

算法 7-18　计算最少用色算法

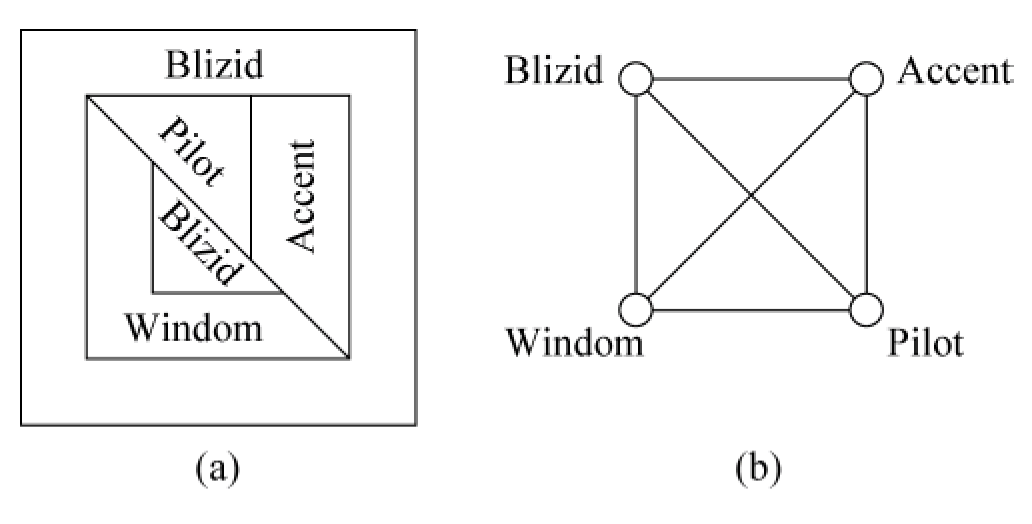


图 7-13　地图转换为图

注意,m-着色问题既非子集树问题,也非排列树问题,所以要用 BACKTRACKITER 过程加以解决。

本题较富挑战性的工作有两个:其一是如何将输入文件中表示成点的序列的区域转换成地图;其二是如何将地图转换成图。

首先,约定存储地图数据的数据结构如图 7-14 所示。

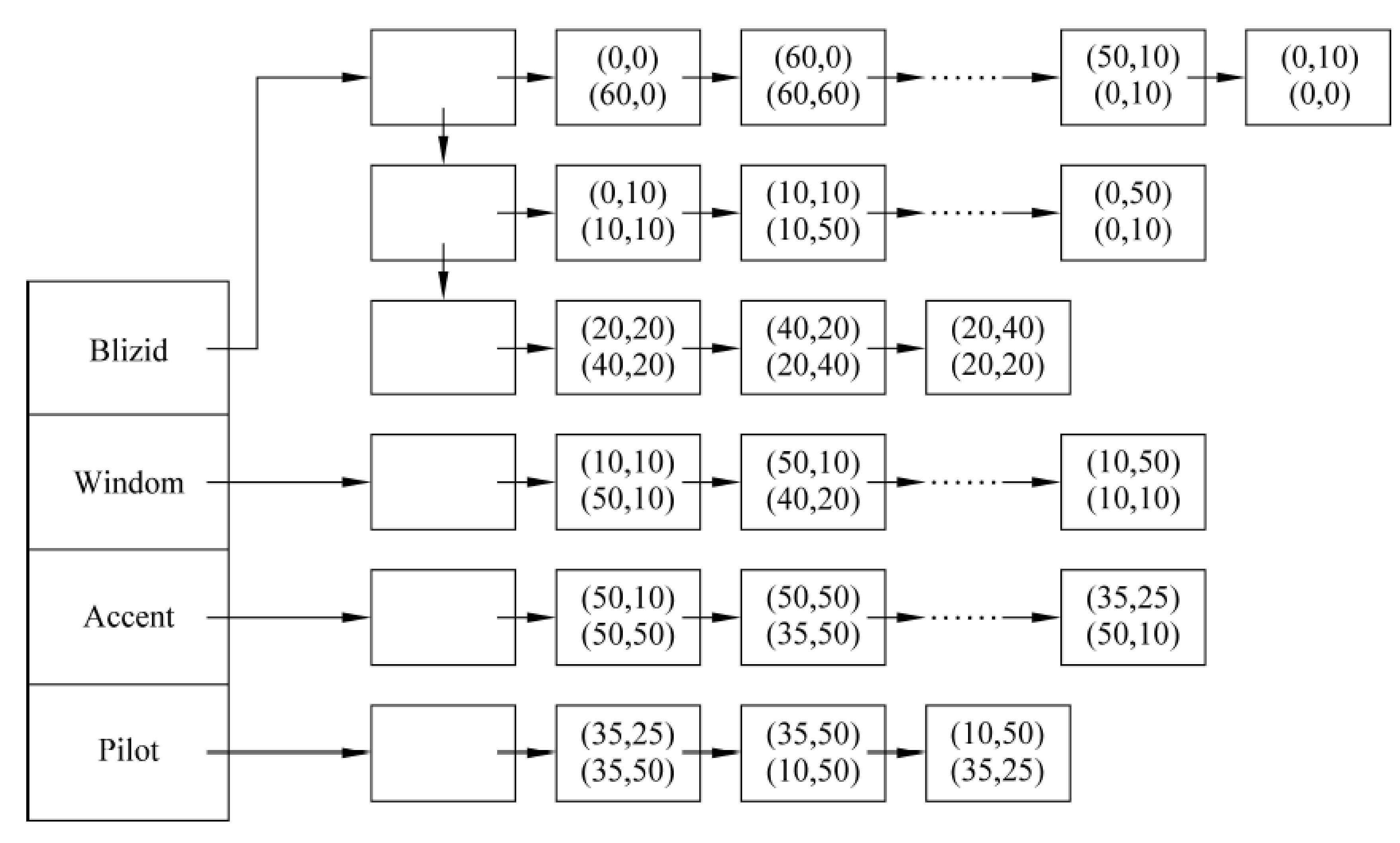


图 7-14　图 7-13(a)中地图的数据表示

也就是说，将一个区域表示成首尾相接的多条线段的序列，存储在一个链表中。将从属于一个国家的所有区域存储在一个链表中。最后将表示一个国家的链表存储为数组的一个元素。

要将上述表示一幅地图的数据结构 *Map* 转换成一个图 *G*（图的邻接矩阵），需对两个不同的国家（分别存储在 *Map* 的第 *i*、*j* 个元素）考察是否有相同的边界。这可利用在第 5 章中讨论的几何算法判断两条分属于不同国家的区域边界上的线段是否平行相交（排除平行相接的情形）。有相交的情形，令 $G[i, j]$为 1，否则令 $G[i,j]$为 0。

2）**程序实现**

```
typedef struct{
  LinkedList *territories;
  char name[21];
}Contry;
int main(){
  FILE *f1=fopen("chap07/ColortheMap/inputdata.txt","r"),      /*输入文件*/
       *f2=fopen("chap07/ColortheMap/outputdata.txt","w");   /*输出文件*/
  assert(f1&&f2);
  fscanf(f1,"%d",&n);
  while(n){                              /*案例中有 n(n>0)个区域*/
      LinkedList *solution=NULL;
      Contry *Map;
      Map= makeMap(f1);                  /*将案例中的 n 个区域的数据构建一个地图*/
      G=map2Graph(Map);                  /*将地图转换为图 G*/
      clrMap(Map,n);free(Map);
      m=0;
      do{                                /*从 m=1 种颜色开始检测是否能对 G 进行 m-着色*/
          ValueSet **OMG;
          m++;
          OMG==makeOMG();                /*构造解空间 Ω*/
          if(solution)                   /*解集非空需清理内存*/
              clrList(solution,clrSolution);
          solution =backtrackiter(OMG, n);/*回溯求解 m-着色问题*/
          clrOMG(OMG,n,NULL);            /*清理解空间 Ω 的内存*/
          free(OMG);
      }while(listEmpty(solution));       /* solution=∅*/
      fprintf(f2,"%d\n",m);
      free(G);
      clrList(solution,clrSolution);free(solution);
      fscanf(f1,"%d",&n);
  }
  return 0;
}
```

程序 7-14　问题的程序实现

对程序 7-14 的说明如下。

(1) 第 1～4 行定义的数据结构是用来表示一个国家的结构体。其中,字符串属性 name 用来表示国家的名称,链表属性 territories 用来存储国家的各个区域。

(2) 第 10～31 行的循环处理每一个数据案例。其中,第 13 行调用函数 makeMap 用输入文件 f1 中的案例数据构造地图 Map。第 14 行调用函数 map2Graph 将 Map 转换成表示图的邻接矩阵 G(按行优先原则存储在一个一维数组中)。第 17～26 行的 **do-while** 循环实现上述伪代码表示 **repeat-until** 循环,计算合法着色的最少色数 m。其中第 23 行调用函数 backtrackiter 回溯法解图 G 的 m-着色问题。第 27 行将算得的最小色数 m 写入输出文件 f2。

(3) 为节省篇幅,函数 makeMap 和 map2Graph 的定义代码没有罗列,读者可打开存储在文件夹 chap07/ Color the Map 中的源文件 ColortheMap.c 研读。第 23 行调用的是本章 7.2.3 节的程序 7-5 定义的实现算法 BACKTRACKITER 的函数 backtrackiter,该函数中调用的 m-着色问题的 isPartial 函数和 isComplete 函数定义在本章 7.2.3 节的程序 7-8 中,读者可参阅。

3. Divisibility

Description

Consider an arbitrary sequence of integers. One can place + or − operators between integers in the sequence, thus deriving different arithmetical expressions that evaluate to different values. Let us, for example, take the sequence: 17, 5, −21, 15. There are eight possible expressions:

17+5+−21+15=16

17+5+−21−15=−14

17+5−−21+15=58

17+5−−21−15=28

17−5+−21+15=6

17−5+−21−15=−24

17−5−−21+15=48

17−5−−21−15=18

We call the sequence of integers divisible by K if + or − operators can be placed between integers in the sequence in such way that resulting value is divisible by K. In the above example, the sequence is divisible by 7 (17+5+−21−15=−14) but is not divisible by 5.

You are to write a program that will determine divisibility of sequence of integers.

Input

The first line of the input file contains two integers, N and K ($1 \leqslant N \leqslant 10000$, $2 \leqslant K \leqslant 100$) separated by a space.

The second line contains a sequence of N integers separated by spaces. Each integer is not greater than 10000 by it's absolute value.

Output

Write to the output file the word "Divisible" if given sequence of integers is divisible by K or "Not divisible" if it's not.

Sample Input

```
4 7
17 5 -21 15
```

Sample Output

```
Divisible
```

1）问题理解

对给定的一组整数 $numbers[1]$，$numbers[2]$，…，$numbers[n]$及整数 K，判断是否存在 $n-1$ 个+、−运算符，将这 n 个整数连接起来的表达式的计算结果能被 K 整除。由于只涉及 $n-1$ 个+、−运算符，所以可以将此问题的解视为向量 $x=\langle x_1, x_2, \cdots, x_{n-1}\rangle$，$x_i=\begin{cases}0 & \text{第 } i \text{ 个运算符为+}\\ 1 & \text{第 } i \text{ 个运算符为−}\end{cases}$（$i=1, 2, \cdots, n-1$）。这是一个子集树问题，可以调用 SUBSET-TREE 过程解决此问题。该问题中部分解$\langle x_1, x_2, \cdots, x_k\rangle$在 $k<n-1$ 时，对应的表达式最终是否被指定的 K 整除是无法决断的，所以只要 $k\leqslant n-1$，就认为是部分合法解。

```
IS-PARTIAL(x₁, x₂, …, xₖ)
if k≤n−1
    then return TRUE
    else return FALSE
```

算法 7-19　IS-PARTIAL 算法

而对于$\langle x_1, x_2, \cdots, x_k\rangle$是否为完整合法解的判断，除了检测 $k=n$ 这个条件之外，还需判断最终的表达式是否能被 K 整除。

```
IS-COMPLETE(x₁, x₂, …, xₖ)
if k=n−1
    then y←numbers[1]
        for i←2 to n
              do if x[i]=0
                    then y←y+numbers[i]
                    else y←y−numbers[i]
    if y|K
        then return TRUE
return FALSE
```

算法 7-20　IS-COMPLETE 算法

2）程序实现

为节省篇幅，此处仅列出实现上述判断合法解过程的代码。解决问题的完整代码存储在文件夹 chap07/ Divisibility 中的源文件 Divisibility. c 中，读者可打开文件研读。

```
1 int n,                              /* 案例中数据个数 */
2     k,                              /* 案例中用来整除表达式的因子 */
3     * numbers;                      /* 存储案例中 n 个数据的数组 */
4 int isPartial(void * x, int k1){    /* 部分合法解判断 */
5   return k1<n;
6}
7 int isComplete(int * x, int k1){    /* 完整合法解判断 */
8   if(k1==n-1){
9       int i,y=numbers[0];
10        for(i=0;i<n;i++){
11            if(x[i]==0)
12                y+=numbers[i+1];
13            else
14                y-=numbers[i+1];
15        }
16        return y%k==0;
17    }
18    return 0;
19 }
```

程序 7-15　问题程序

第 1～3 行定义的全局变量 n、k 和 numbers 分别用来表示一个案例中的数据个数、整除表达式的因子和存放 n 个数据的数组。第 4～6 行定义的函数 isPartial 和第 7～19 行定义的函数 isComplete 实现了上述的同名伪代码过程。代码结构几乎与伪代码过程一致。

7.4　用回溯算法解决组合优化问题

有了在组合问题的解空间中进行回溯搜索合法解的算法,本节来讨论如何用这一方法解决组合优化问题。

7.4.1　组合优化问题

如果组合问题中的每一个可能解均对应一个数值,希望寻求合法解中对应数值最小的或最大的,则构成一个**组合优化问题**。组合优化问题中每个可能解对应的数值称为该解的**目标值**,目标值最小或最大的解称为该问题的**最优解**。下面通过几个经典例子来说明组合优化问题。

1. 0-1 背包问题

设有 n 种物品,第 i 种物品的质量为 w_i,价值为 v_i,$i=1, 2, \cdots, n$。给定一个背包,最多能装质量为 C 的物品。第 i 种物品或放入包中,或留在包外。问将哪些物品放到包内能使得包中所装物品总价值最大?其中,w_i、v_i 都是正整数,$i=1, 2, \cdots, n$。C 也是正整数。若

用一个向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$来表示此问题的解。其中 $x_i=\begin{cases}1 & i\text{ 号物品放入包中}\\0 & i\text{ 号物品未放入包中}\end{cases}$。则 0-1 背包问题可形式化地描述为如下。

输入：集合 $W=\{w_1, w_2, \cdots, w_n\}$，$V=\{v_1, v_2, \cdots, v_n\}$，数值 C。

输出：向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$，$x_i\in\{0,1\}$，$1\leqslant i\leqslant n$，使得 $\sum x_i w_i\leqslant C$，且 $\sum x_i v_i$ 最大。

这是一个典型的组合优化问题。所有 2^n 个向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$，$x_i\in\{0,1\}$，$1\leqslant i\leqslant n$，构成问题的解空间。可行解受 $\sum x_i w_i\leqslant C$ 限制且以 $\sum x_i v_i$ 为目标值。目的是求出可行解中的最优解——目标值最大者。

2. 活动选择问题

回顾在 7.3.1 节中讨论过的相容活动问题。n 个具有各自活动时间区间$[s_i, f_i)$的活动 a_i，$i=1, 2, \cdots, n$，竞争一个资源。若$[s_i, f_i)\cap[s_j, f_j)=\varnothing$，则称 a_i、a_j 相容。希望寻求活动集合 $A=\{a_1, a_2, \cdots, a_n\}$的一个最大相容子集合。若将该问题的解表示成向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$，其中 $x_i=\begin{cases}1 & \text{选择 } i\text{ 号活动}\\0 & \text{不选择 } i\text{ 号活动}\end{cases}$，则活动选择问题可形式化地表示为如下。

输入：按完成时间排好序的活动开始时间数组 s、完成时间数组 f，且活动按完成时间的升序排列，即 $f_1\leqslant f_2\leqslant\cdots\leqslant f_n$。

输出：表示一个最大的相容活动组的向量$\langle x_1, x_2, \cdots, x_n\rangle$，其中

$$x_i=\begin{cases}1 & \text{第 } i\text{ 个活动在此最大相容活动组中}\\0 & \text{否则}\end{cases},\quad i=1, 2, \cdots, n$$

这也是一个组合优化问题。每一个可行解$\langle x_1, x_2, \cdots, x_n\rangle$都对应一个目标值：其中不为 0 的分量个数。最优解是目标值最大的解。

3. 旅行商问题

再来看一个例子——旅行商问题：商人从 n 个城市中的一个出发，希望走遍每个城市且每个城市只经过一次，回到出发的城市。如果有这样的路径，要求找到里程最短的。这实际上是在要求最短的 Hamilton 回路。问题形式化为如下。

输入：带权无向图 $G=\langle V, E\rangle$，其中 $V=\{1, 2, \cdots, n\}$，$E\subseteq V\times V$；权函数 $w: E\rightarrow\mathbf{R}$。起始顶点 s。

输出：如果 G 中存在 Hamilton 回路，则输出权值最小的。

这也是一个经典的组合优化问题，每一个可行解是通过图 G 中每个顶点一次的简单回路，它可对应一个向量$\langle x_1, x_2, \cdots, x_n\rangle$，其分量构成 $1, 2, \cdots, n$ 的一个排列满足 $x_1=s$，$(x_n, s)\in E[G]$。每个可行解对应一个目标值：该回路的长度(每条边的权值之和)，最优解的目标值最小。旅行商问题的一个例子如图 7-15 所示。

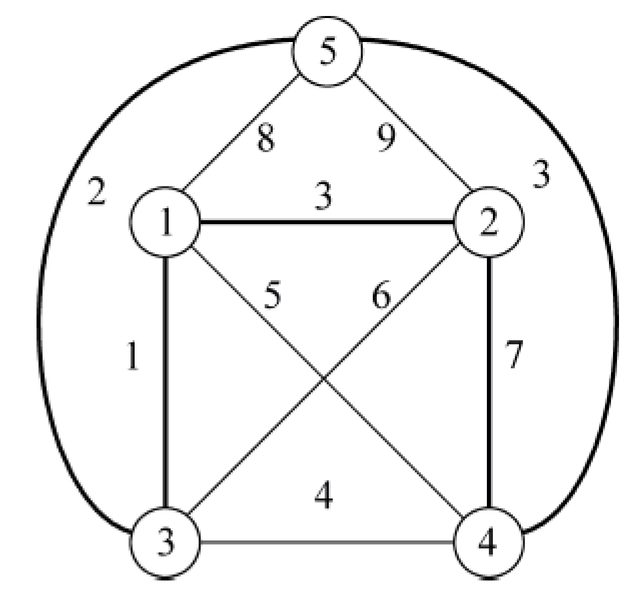


图 7-15　一个带权无向图的最短 Hamilton 回路

7.4.2 用回溯策略解决组合优化问题

从 7.4.1 节的 3 个例子可见,对于组合优化问题可以先将其视为组合问题,利用回溯算法得出所有可行解,然后计算每个可行解的目标值,最终得到最优解。然而,如果跟踪当前最优解的目标值 *most*(对最小优化问题初始化为∞,最大优化问题初始化为$-\infty$),可在回溯搜索过程中,对部分可行解 x 将其目标值 *obj* 与 *most* 比较,若不可能比之更优则放弃对该分支的搜索。对完整可行解 x,若其目标值 *obj* 比 *most* 更优,则改写 *most* 且记录当前最优解。这样可以提高搜索效率。

于是,对判断组合优化问题的部分可行解的过程应该具有如下结构。

```
IS-PARTIAL(x1, x2, …, xk)
if <x1, x2, …, xk>不合法
    then return false
obj←OBJ-VALUE(x1, x2, …, xk)
if obj 不可能比 most 更优
    then return false
return true
```

算法 7-21　判断组合优化问题的部分可行解

而对判断组合优化问题的完整可行解的过程应该具有如下结构。

```
IS-COMPLETE(x1, x2, …, xk)
if <x1, x2, …, xk>不是完整合法
    then return false
if obj 不比 most 更优
    then return false
most←obj
return true
```

算法 7-22　解决组合优化问题的回溯算法

对具体问题设计算法 7-22 中的过程,调用回溯算法(包括子集树和排列树算法)就能解决该问题。注意在回溯算法中,IS-COMPLETE 的调用总是在 IS-PARTIAL 之后,所以 *obj* 的计算仅在 IS-PARTIAL 进行,在 IS-COMPLETE 中直接引用就行了。下面就上述的 3 个例子说明这一思想的运用。

1. 0-1 背包问题

对于 0-1 背包问题,其问题模型属于子集树问题,搜索树形象见图 7-16。假定 n 件物品的质量和价值数据存储于数组 w 和 v 中。由于最优解是目标值最大的可行解,所以将当前最优解目标值 *most* 初始化为$-\infty$。记所有物品的总价值 $total=\sum_{i=1}^{n} v_i$。对部分向量 $\langle x_1, x_2, \cdots, x_k\rangle$计算质量 $weight=\sum_{i}^{k} x_i w_i$,若可行($weight \leqslant C$),计算出其目标值

$obj=\sum_{i}^{k} x_i v_i$。比较 $obj+\sum_{i=k+1}^{n} v_i$ 与 $most$ 的大小，前者表示当前部分解加上第 $k+1$，…，n 的所有物品的价值，后者是到目前为止得到的最优解的价值，若前者不超过后者，意味着 x_{k+1}，…，x_n 无论如何取值，都不可能比当前的最优解更优，故可放弃。若前者大于后者，意味着进一步探索有可能得到比当前更好的解。对于完整的可行解 $<x_1, x_2, \cdots, x_n>$，若 $obj>most$ 意味着比当前最优解更好，跟踪此解。将此思想写成伪代码过程如下。

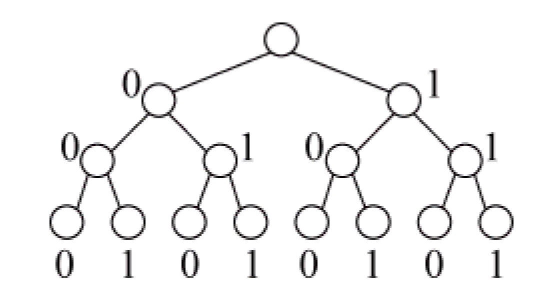


图 7-16　具有 3 个物件的 0-1 背包问题的搜索空间

```
IS-PARTIAL(x1, x2, …, xk)
obj←weight←0, d←total
for i←1 to k
    do weight←weight+ xi · wi
       if weight>C
          then return false
       obj← obj+ xi · vi
       d←d− vi
if obj+d≤most
    then return false
return true
IS-COMPLETE(x1, x2, …, xk)
if k<n
    then return false
if obj≤most
  then return false
most←obj
return true
```

算法 7-23　判断 0-1 背包问题中部分可行解与最优解的过程

IS-PARTIAL 过程的运行时间是 $O(n)$，而 IS-COMPLETE 过程的运行时间是 $O(1)$。由于 0-1 背包问题是子集树问题，所以可以调用 SUBSET-TREE 过程解决该问题。

2. 活动选择问题

活动选择问题也是子集树问题。与 0-1 背包问题相似，最优解也是目标值最大的可行解。所以，当前最优解目标值 $most$ 也初始化为 $-\infty$。对于部分解向量 $<x_1, x_2, \cdots, x_k>$，计算目标值 obj 为 $\sum_{j=1}^{k} x_j$，以及最后一个不为 0 的分量下标 i。若 $x_k\neq 0$ 且 $f_i\leqslant s_k$，比较 $obj+n-k$ 与 $most$ 的大小。前者表示当前部分解加上后面所有活动的数量，后者为当前最优解中活动数量。若前者不超过后者，意味着当前部分解不可能扩展成最优解，应放弃进一步探索。对完整可行解，只要 $obj>most$ 即为新的当前最优解，跟踪 $most$。写成伪代码过程如下。

```
IS-PARTIAL(x1, x2, …, xk)
```

```
obj ← j ← 0
for j ← 1 to k−1
    do if x_j = 1
        then obj ← obj + 1
             i ← j
if x_k = 1
    then obj ← obj + 1
        if i ≥ 1 and f_i > s_k
            then return false
if obj < most − n + k
    then return false
return true
IS-COMPLETE(x_1, x_2, …, x_k)
if k < n or obj ≤ most
    then return false
most ← obj
return true
```

算法 7-24　判断活动选择问题中部分可行解与最优解的过程

与算法 7-23 一样,IS-PARTIAL 过程的运行时间是 $O(n)$,而 IS-COMPLETE 过程的运行时间是 $O(1)$。

3. 旅行商问题

旅行商问题是在 Hamilton 回路问题的可行解中找出最优解,所以适合于排列树搜索。与 0-1 背包问题和活动选择问题不同,旅行商问题的最优解是目标值最小的可行解。所以将当前最优解目标值 $most$ 初始化为∞。记 $maxedge=G$ 中以 s 为端点的最长边的权。对于部分解向量$<x_1, x_2, \cdots, x_k>$,若 $s, x_1, x_2, \cdots, x_k$ 构成图 G 的一条简单路径,计算目标值 $obj=w(s, x_1)+\sum_{i=1}^{k-1} w(x_i, x_{i+1})$,比较 obj 与 $most\text{-}maxedge$ 的大小。前者是当前部分解的目标值,后者不超过当前最优解的目标值去除回路中封口边后的权值,若前者不小于后者,则$<x_1, x_2, \cdots, x_k>$不可能扩展成新的更优解,故放弃进一步探索。对于完整解$<x_1, x_2, \cdots, x_{n-1}>$,若 $obj<most$ 则构成新的最优解,跟踪 $most$。写成伪代码过程如下。

```
IS-PARTIAL(x_1, x_2, …, x_k)
if (s, x_1) ∉ E[G]
    then return false
obj ← w(s, x_1)
for i ← 1 to k−1
    do if (x_i, x_{i+1}) ∉ E[G]
           then return false
           obj ← obj + w(x_i, x_{i+1})
        if obj ≥ most-maxedge
           then return false
return true
```

```
IS-COMPLETE(x_1, x_2, …, x_k)
if k<n−1 or (x_k, s)∉E[G]obj≤most
    then return false
obj←obj+w(x_k, s)
if obj≥most
    then return false
most←obj
return true
```

算法 7-25　判断旅行商问题中部分可行解与最优解的过程

与前两个算法一样，IS-PARTIAL 过程的运行时间是 $O(n)$，而 IS-COMPLETE 过程的运行时间是 $O(1)$。

将算法 7-23～算法 7-25 实现为 C 函数，并调用合适的回溯搜索函数解决 0-1 背包问题、活动选择问题和旅行商问题的程序分别写入 btrack 文件夹的源文件 knapsack.c、activeselect.c和 tsp.c 中。为节省篇幅计，此处不再罗列，读者可打开文件研读。

7.4.3　应用

1. Calculate A Route

Sichuan province is saturated with many beautiful, stirring and mysterious stories about the Three Kingdoms. Ancient Jianmen Trail is especially a good resort with both beautiful scenery and culture. The most exiting story, among many others, is *Riding Alone for Thousands of Miles*, an interesting description about Guan Yu, a senior general, who, upon hearing about the situation of his sworn brother Liu Bei, give up all the excellent offers by Cao Cao and embarked on the road to look for Liu. Broadsword on back, followed by Liu's family, riding on his treasured horse, Guan started from Xudu, and finally found Liu in Gucheng after overcoming all the difficulties in his way.

Now, your task is to design a new route for Guan Yu to find his brother. Suppose that Guan Yu knew all the connection between the passes from Xudo to Gucheng, and the probability to defeat the guards on each pass to go through safely. Please help Guan Yu find a successful route from Xudo to Gucheng with a maximum of probability. Guan Yu cannot fo to each pass twice.

Input

The first line of the input contains an integer T ($1 \leqslant T \leqslant 50$). T refers to the number of test cases followed.

For each test case, the first line contains only one integer N ($3 \leqslant N \leqslant 1000$). The following N lines are a matrix representing the connections among the N-2 passes from Xudo to Gucheng. If the number of the ith row and the jth column is 1, this means that there is a connection from i to j. Otherwise, the number of 0 means no connection from i to j. The last line of each test case contains N-2 real numbers separated by a single blank,

and each real number represents the probability P ($0 \leqslant P \leqslant 1$) of Guan Yu going through the pass safely.

Output

For each test case you should output one line containing only one real number representing the maximum probability of Guan Yu arrive in Gucheng safely. The real number is rounded to four decimal digits. If the result is less than 0.0001, then output "Cannot reach!". The format of output is illustrated in Sample output.

Sample Input

```
3
3
0 1 0
0 0 1
0 0 0
0.5
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
0.01 0.001
6
0 1 1 0 0 0
0 0 1 1 0 0
0 0 0 1 1 0
0 0 1 0 1 1
0 0 0 0 0 1
0 0 0 0 0 0
0.2 0.1 0.6 0.3
```

Output for the Sample Input

```
Case 1: 0.5000
Case 2: Cannot reach!
Case 3: 0.1200
```

1) 问题理解

关羽谢绝了曹操的优厚待遇,护送刘备夫人从许都出发,过重关,越千里,到古城寻找刘备。对关羽而言,如果从许都出发所到各关口能破此关的概率是已知的,要计算关羽从许都出发最有把握到达到古城的路线。这个问题中有一个有向图 $G=\langle V, E\rangle$,并给定图中的源顶点 s(许都)和目标顶点 d(古城),图中顶点 $u\in V-\{s, d\}$,均对应一个非负实数值 $0\leqslant p(u)\leqslant 1$(输入范例对应的图如图 7-17 所示,顶点的权值标示在其旁边,粗箭头指示出一条最优路径)。为统一操作,补充定义 $p(s)=p(d)=1$。目的是在 G 中找一条从 s 到 d 的路径,使得该条路径上的各顶点对应的权值之积为最大。为表述方便,设 $V=\{1, 2, \cdots, n\}$,并设

$s=1$，$d=n$。则从 1 到 n 的一条路径为<1，x_1，…，x_{n-1}，n>，路径上所有顶点对应的权值之积为 $\prod_{i=1}^{n-1} p(x_i)$。假定用邻接矩阵来表示图 G，问题的形式化表述如下。

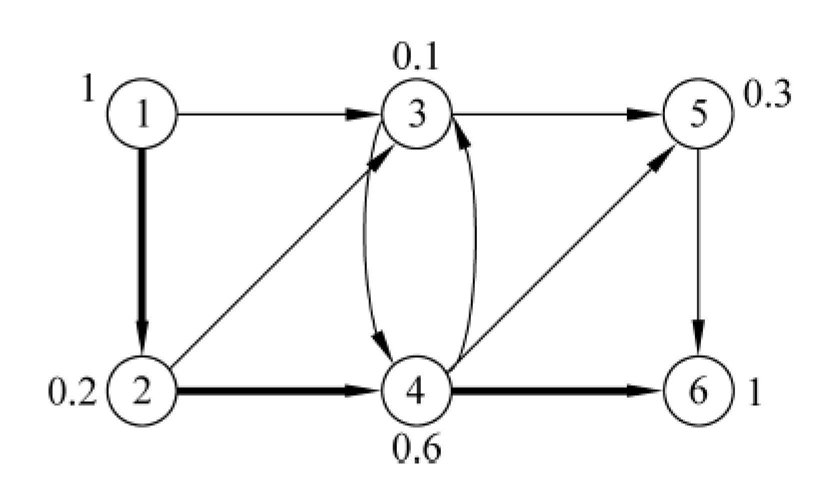
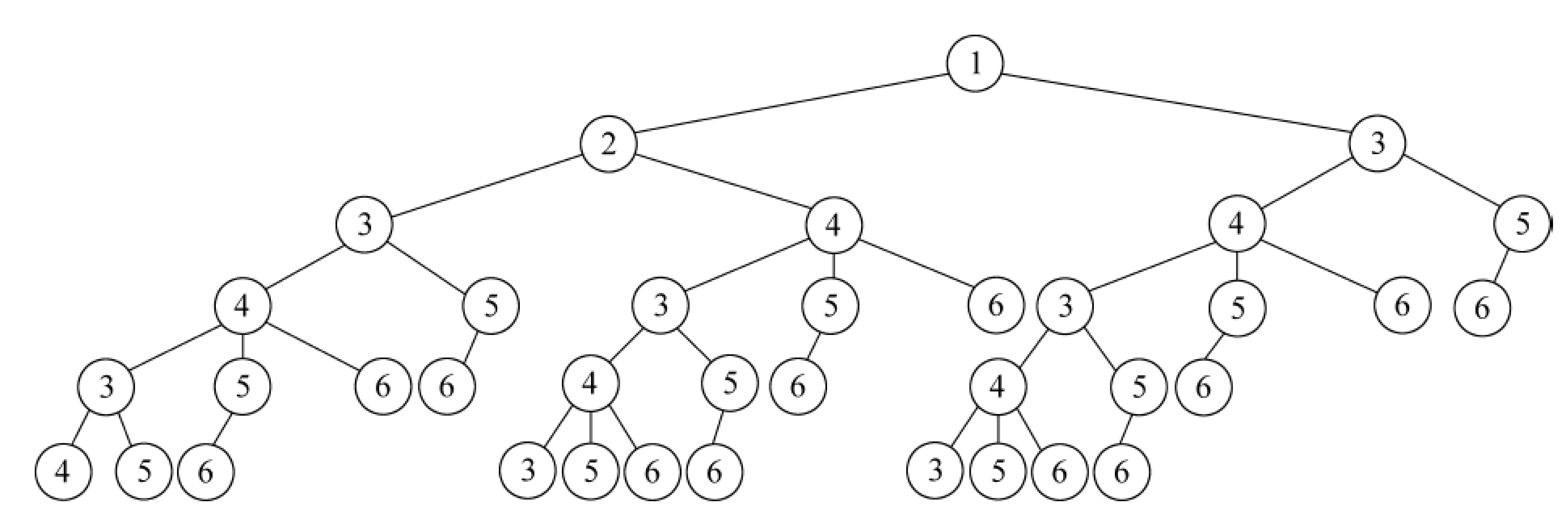


图 7-17　输入范例 3 中的有向图 G

输入：表示有向图 G 的邻接矩阵 $G_{n\times n}$ 和每个顶点的权值数组 p。

输出：G 中从 1 到 n 的路径<1，x_1，…，x_{n-1}，n>，使得 $\prod_{i=1}^{n-1} p(x_i)$ 最大，该积小于0.0001，或根本就不存在 1 到 n 的路径，则输出"Cannot reach!"；否则，输出此积。

这是一个组合优化问题。从顶点 1 出发到达任一顶点的路径构成搜索空间。从 1 到 n 的路径为可行解。每个可能解<1，x_1，…，x_{n-1}，n>以 $\prod_{i=1}^{n-1} p(x_i)$ 作为其目标值，目的是找到目标值最大的可行解。可以将此问题的搜索空间组织成一棵高度不超过 $n-1$ 的根树。根结点对应于最优路径起点 1。第 1 层对应于最优路径上从 1 出发第 1 个来到的顶点的各种可能的选择，第 2 层上的结点对应于最优路径上从 1 出发第 2 站的所有可能的选择，……。输入范例 3 的搜索树，如图 7-18 所示。


图 7-18　输入范例 3 的搜索树

和旅行商问题类似，本问题寻求 G 中从 $s(s=1)$到 n 的一条无重复顶点的路径，不同的是不必经过 G 中的每一个顶点，并且要求这条路径上的顶点权重之积最大。可以用解决排列树模型的 PERMUT-TREE 过程加以解决。

2）**算法的伪代码描述**

用 $s=1$，$t=n$ 表示起点和终点，max 来跟踪最优解的目标值，初始时为$-\infty$。为判断部分解< x_1，x_2，…，x_k>是否合法，首先需要判断 s，x_1，x_2，…，x_k 是否构成 G 中一条路径。若是，则需要计算这条路径权 $obj=\prod_{i=1}^{k} p(x_i)$，并检测 obj 是否大于 max。写成过程如下。

```
IS-PARTIAL(x1, x2, …, xk)
1 if (s, x1) ∉ E[G]
2     then return false
```

```
obj← p(x1)
for i←1 to k−1
    do if (xi, xi+1)∉E[G]
            then return false
        obj← obj×p(xi+1)
        if obj≤max
            then return false
return true
```

算法 7-26 IS-PARTIAL 算法

由于判断< x_1, x_2, …, x_k >是否为完整合法解后于判断其是否为部分合法解,所以只需检测是否(x_k, n)∈$E[G]$,若是,改写 max(找到了新的更优的解,其目标值 obj 大于 max)。写成伪代码过程如下。

```
IS-COMPLETE(x1, x2, …, xk)
if (x[k], n) ∉E[G]
    then return false
max←obj
return true
```

算法 7-27 判断是否为部分合法解

3) **程序实现**

为节省篇幅,此处仅列出实现上述伪代码过程的函数,解决该问题完整的程序代码存储在文件夹 chap07/Calculate A Route 中的源文件 CalculateARoute.c 中,读者可打开此文件研读。

```
double *p,                          /*关隘过关概率*/
       max,                         /*当前最优解的目标值*/
       obj;                         /*当前可行解的目标值*/
int n,                              /*图 G 的顶点个数*/
    s=1,                            /*起点*/
    *G,                             /*图 G 的邻接矩阵*/
    *origin;                        /*去除起点、终点后的顶点初始排列*/
int isPartial(int *x, int k){       /*部分可行解判断*/
    int i;
    if(G[(s-1)*n+x[0]-1]==0)
        return 0;
    obj=p[x[0]-1];
    for(i=0;i<k;i++){
        if( G[(x[i]-1)*n+x[i+1]-1]==0)
            return 0;
        obj*=p[x[i+1]-1];
        if(obj<=max)
            return 0;
    }
```

```
20      return 1;
21 }
22 int isComplete(int * x, int k){
23      if (G[(x[k]-1) * n+n-1]==0)
24            return 0;
25      if(obj<=max)
26            return 0;
27      max=obj;
28      return 1;
29 }
```

程序 7-16　问题程序实现

第 1 行定义的全局指针变量 p 用来指向存储各个关隘的过关概率的数组。第 2 行和第 3 行定义的全局变量 max 和 obj 用来表示当前最优解的目标值和当前可行解的目标值。第 4 行和第 5 行定义的变量 n 和 s 分别用来表示图 G 中的顶点个数及起点(顶点 1)。第 6 行定义的指针变量 G 用来指向图的邻接矩阵(按行优先原则存储在一维数组中)。第 7 行定义的指针变量 origin 用来指向去除了起点 s=1 和终点 n 以外 G 中所有顶点初始顺序的数组。

第 8～21 行定义的函数 isPartial 和第 22～29 行定义的函数 isComplete 实现上述同名算法过程，代码结构与伪代码的几乎相同。

2. Graph Coloring

Description

You are to write a program that tries to find an optimal coloring for a given graph. Colors are applied to the nodes of the graph and the only available colors are black and white. The coloring of the graph is called optimal if a maximum of nodes is black. The coloring is restricted by the rule that no two connected nodes may be black.

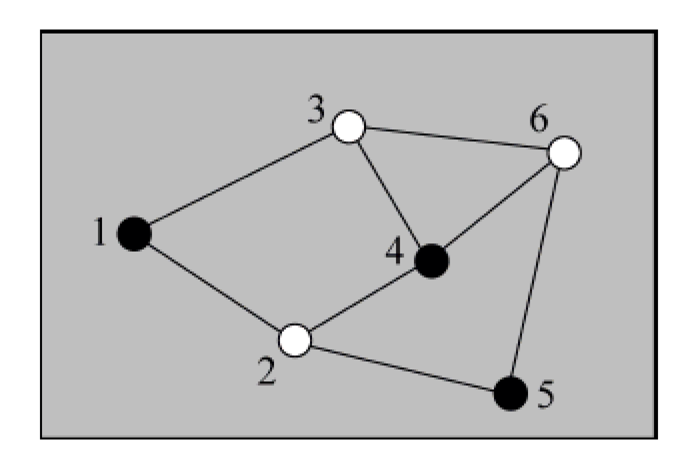


图 7-19　An optimal graph with three black nodes

Input

The graph is given as a set of nodes denoted by numbers 1..n, $n \leqslant 100$, and a set of undirected edges denoted by pairs of node numbers (n_1, n_2), $n_1 \neq n_2$. The input file contains m graphs. The number m is given on the first line. The first line of each graph contains n and k, the number of nodes and the number of edges, respectively. The following k lines contain the edges given by a pair of node numbers, which are separated

by a space.

Output

The output should consists of $2m$ lines, two lines for each graph found in the input file. The first line of should contain the maximum number of nodes that can be colored black in the graph. The second line should contain one possible optimal coloring. It is given by the list of black nodes, separated by a blank.

Sample Input

```
1
6 8
1 2
1 3
2 4
2 5
3 4
3 6
4 6
5 6
```

Sample Output

```
3
1 4 5
```

1) **问题理解**

对输入的图 $G=\langle V, E\rangle$,$V=\{1, 2, \cdots, n\}$,$E\subseteq V\times V$,用黑白两色对顶点涂色,要求两个相邻顶点不能同涂黑色。寻求最佳涂色方案:黑色顶点最多。该问题的解可视为向量 $x=\langle x_1, x_2, \cdots, x_n\rangle$,$x_i=\begin{cases}0 & \text{第 } i \text{ 个顶点涂白色}\\1 & \text{第 } i \text{ 个顶点涂黑色}\end{cases}$$(i=1, 2, \cdots, n)$,因此可以用基于子集树模型的 SUBSET-TREE 过程加以解决。问题中判别$\langle x_1, x_2, \cdots, x_k\rangle$是否为合法部分解的伪代码过程如下。

```
IS-PARTIAL(x1, x2, …, xk)
obj←0
for i←1 to k-1
    do if (xi, xk)∈E[G] and xi=1 and xk=1 ▷ xi、xk 相邻且均为黑色
            then return false
       obj← obj+xi
if obj+n-k≤max                      ▷即使加上后面的 n-k 个黑色顶点均无法赶上当前的最优解
    then return false
return true
```

算法 7-28 判断是否为合法解的算法

对部分合法解$\langle x_1, x_2, \cdots, x_k\rangle$判断其是否为完整合法解的过程如下。

IS-COMPLETE($x_1, x_2, \cdots, x_k$)

```
if k<n
    then return false
max←obj
return true
```

算法 7-29　判断部分合法解是否是完整合法解的算法

2）**程序实现**

为节省篇幅，此处仅列出实现上述判断合法解过程的函数。解决本问题完整的程序代码存储在文件夹 chap07/ Graph Coloring 中的源文件 Graph Coloring. c 中，读者可打开该文件研读。

```
1 int *G,
2   n,
3   max=0,
4   obj;
5 int isPartial(int* x, int k){                 /* 部分合法解判断 */
6   int i;
7   for(i=0,obj=0;i<k;i++){
8       obj+=x[i];
9       if(G[i*n+k]&&x[i]==1&&x[k]==1)
10          return 0;
11   }
12   if(obj+n-k<=max)
13       return 0;
14   return 1;
15 }
16 int isComplete(int* x, int k){                /* 完整合法解判断 */
17   int i;
18   if(k<n-1)
19       return 0;
20   max=obj;
21   return 1;
22 }
```

程序 7-17　Graph Coloring 源程序

第 1～4 行定义的指针变量 G 指向图的邻接矩阵，变量 n 表示图 G 的顶点数，变量 obj 和 max 分别表示当前解的目标值和当前最优解的目标值。

第 5～15 行定义的函数 isPartial 和第 16～22 行定义的函数 isComplete 实现的是上述的同名过程代码结构，它们几乎完全一致。

3. Communication System

Description

We have received an order from Pizoor Communications Inc. for a special communication system. The system consists of several devices. For each device, we are

free to choose from several manufacturers. Same devices from two manufacturers differ in their maximum bandwidths and prices.

By overall bandwidth (B) we mean the minimum of the bandwidths of the chosen devices in the communication system and the total price (P) is the sum of the prices of all chosen devices. Our goal is to choose a manufacturer for each device to maximize B/P.

Input

The first line of the input file contains a single integer t ($1 \leqslant t \leqslant 10$), the number of test cases, followed by the input data for each test case. Each test case starts with a line containing a single integer n ($1 \leqslant n \leqslant 100$), the number of devices in the communication system, followed by n lines in the following format: the i-th line ($1 \leqslant i \leqslant n$) starts with m_i ($1 \leqslant m_i \leqslant 100$), the number of manufacturers for the i-th device, followed by m_i pairs of positive integers in the same line, each indicating the bandwidth and the price of the device respectively, corresponding to a manufacturer.

Output

Your program should produce a single line for each test case containing a single number which is the maximum possible B/P for the test case. Round the numbers in the output to 3 digits after decimal point.

Sample Input

```
1
3
3 100 25 150 35 80 25
2 120 80 155 40
2 100 100 120 110
```

Sample Output

```
0.649
```

1) **问题理解**

一个网络系统由 n 个设备组成,第 $i(1 \leqslant i \leqslant n)$个设备有 m_i 个供应商,其中第 $j(1 \leqslant j \leqslant m_i)$个供应商提供的设备带宽为 b_{ij}(单位:bps),价格为 p_{ij}(单位:元)。对每一个设备选择一个供应商,使得 n 个设备的 B/P 最大。其中,B 表示 n 个设备中带宽最小者的带宽,P 表示所选的这 n 个设备价格之和。

本问题可视为解空间 $\Omega = X_1 \times X_2 \times \cdots \times X_n$,其中 $X_i = \{1, 2, \cdots, m_i\}(i=1, 2, \cdots, n)$的组合优化问题。解向量 $x = <x_1, x_2, \cdots, x_n>$满足 $B = \min\{b_{1x_1}, b_{2x_2}, \cdots, b_{nx_n}\}$,$P = \sum_{i=1}^{n} p_{ix_i}$,$B/P$ 最大。显然此问题既非子集树类型,亦非排列树类型,所以需要调用 BACKTRACKITER 过程求解。

由于本问题中解的目标值必须在 n 个分量确定后才能算得,所以部分解$<x_1, x_2, \cdots, x_k>$合法的判断只需检测 $k \leqslant n$。而当 $k = n$ 时,需要计算出 $B = \min\{b_{1x_1}, b_{2x_2}, \cdots, b_{nx_n}\}$及

$P=\sum_{i=1}^{n} p_{ix_i}$ 的商 B/P 作为解的目标值 obj，并与 max 比较，决定是否得到新的最优解。写成伪代码过程如下。

```
IS-PARTIAL(x_1, x_2, …, x_k)
if k≤n
    then return true
return false
IS-COMPLETE(x_1, x_2, …, x_k)
if k<n
    then return false
B←min {b_{1x_1}, b_{2x_2}, …, b_{nx_n}}
P← ∑_{i=1}^{n} p_{ix_i}
obj←B/P
if obj<max
    then return false
max←obj
return true
```

算法 7-30　最优解算法

2）程序实现

此处仅列出实现上述算法过程的函数代码，完整程序代码存储在文件夹 chap07/Communication System 中的源文件 CommunicationSystem.c 中，读者可打开此文件研读。

```
int n, *m=NULL, **b=NULL, **p=NULL;
int isPartial(int *x, int k){
  return k<n;
}
int isComplete(int *x, int k){
  int minb=INT_MAX,sump=0,i;
  if(k<n-1)
     return 0;
  for(i=0;i<n;i++){
      sump+=p[i][x[i]];
      if(b[i][x[i]]<minb)
         minb=b[i][x[i]];
  }
  obj=(double)minb/sump;
  if(obj<=max)
      return 0;
  max=obj;
  return 1;
}
```

程序 7-18　问题程序实现

第 1 行中定义的变量 n 表示设备数,指针变量 m 指向存储各设备的供应商数的数组。指针变量 b 和 p 分别指向存储各设备的各供应商提供设备的带宽数组和价格数组。

第 2～4 行定义的函数 isPartial 和第 5～19 行定义的函数 isComplete 实现的是上述的同名算法过程,代码结构几乎一致。

第 8 章　动态规划策略

在第 7 章中，介绍了解决组合优化问题的直接方法是"回溯算法"：搜索组织成根树的问题的解空间，并计算可行解的目标值，从中选取具有最优值的解。然而，很多组合优化问题的解空间是无限的。即使问题的解空间是有限集合，其中的元素个数很可能是巨大的。此时，回溯算法将会陷入一个"时间黑洞"。如何有效地解决组合优化问题是计算机技术的一个既传统又现实的研究应用领域。研究思路之一是发掘问题本身的特性，针对特性利用特殊的算法策略设计有效算法。本章中针对一类具有"最优子结构"性质及"子问题重叠"性质的组合优化问题来讨论一种解决问题的算法设计策略——"动态规划"。

1. 最优子结构

最优子结构特性是指问题的最优解包含的子问题的解相对于子问题而言也是最优的。例如，考虑如下最短路径问题。

已知有向图 $G=<V,E>$ 和顶点 u、$v\in V$。找出从 u 到 v 的由最少的边构成的路径（也是 G 中从 u 到 v、经过最少中间顶点的路径。这是通信网络中两结点间的数据传送经过最少中间结点转发的信道问题的模型）。显然，这样的路径必为简单路径，这是因为从路径上删除一个环将使得路径包含的边更少。假定从 u 到 v 存在这样的一条最短路径 $p=(v_1,v_2,\cdots,v_n)$，其中 $v_1=u$，$v_n=v$。对于任意 $1<i<n$，设 $p_1=(v_1,\cdots,v_i)$ 和 $p_2=(v_i,\cdots,v_n)$，则 p 的长度等于 p_1 的长度与 p_2 的长度之和。人们断言，p_1 和 p_2 分别是 G 中 u 到 v_i 的最短路径和 v_i 到 v 的最短路径。下面，说明 p_1 是 G 中 u 到 v_i 的最短路径，对于 p_2，得相同结论，读者可仿照说明。假定 p_1 不是 G 中 u 到 v_i 的最短路径，设 p_1' 是 G 中 u 到 v_i 的一条最短路径，当然有 p_1' 的长度小于 p_1 的长度。将 p_1' 和 p_2 相连（p_1' 的末尾顶点与 p_2 的首顶点都是 v_i），得到一条新的从 u 到 v 的路径 p' 且其长度为 p_1' 的长度与 p_2 的长度之和。这样，就可推得 p' 的长度小于 p 的长度。这与 p 是 G 中 u 到 v 的一条最短路径矛盾。

组合优化问题的最优子结构揭示了该问题的最优解与其子问题的最优解之间的关系。借助这个关系，可以考虑如何通过解决子问题来达到解决问题的本身。需要注意的是并非所有的组合优化问题都具有最优子结构特性。例如，同样在有向图 $G=<V,E>$ 中考虑从 u 到 v 的包含最多条边的简单路径（也是 G 中从 u 到 v，经过最多中间顶点的简单路径。这是旅游时经过最多景点的游玩路径问题的模型）。这是一个组合优化问题但它不具有最优子结构特征，我们用一个例子加以说明，如图 8-1 所示。

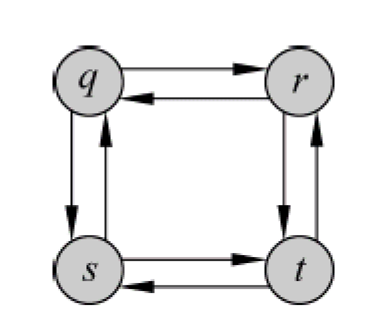


图 8-1　有向图顶点间的路径

在图 8-1 中，路径 $q\to r\to t$ 是 q 到 t 的最长简单路径，但子路径 $q\to r$ 却不是 q 到 r 的最长简单路径，子路径 $r\to t$ 也不是 r 到 t 的最长简单路径。

2. 子问题重叠

由于问题具有最优子结构特性，很自然地想到用分治算法：递归解子问题的最优解，然

后合并成问题的最优解。然而,问题往往没有那么简单。首先,最优子结构是在已知问题最优解的前提下说明其与子问题最优解之间的关系。因此,要由子问题的最优解得到问题的最优解,需要把问题分解成各种可能的子问题组合情形进行考察。在对各种可能的子问题分解情形的考察过程中,有可能对低层的子问题反复进行计算。此时,说该最优化问题具有**重叠子问题**特性。

针对具有上述两个特征的组合优化问题,动态规划算法通常需要做如下三步工作。

(1) 利用最优子结构定义一个关于解的目标值的递归方程。鉴于子问题的重叠性,如果自顶向下地用递归技术解每一个遇到的子问题,则会反复解低层的子问题。

(2) 因此,动态规划以自底向上地对每个新产生的子问题仅解一次且将其解保存在一个表格中,需要时可以在表中查找,且能在常数时间内完成查找。

(3) 根据计算出的最优解的值构造对应的最优解。

本章将运用动态规划策略解决几个经典的组合优化问题。

8.1 组装线调度问题

8.1.1 问题描述

先用动态规划策略解决一个生产问题。某汽车公司在一个工厂里生产汽车,这个工厂有两条组装线,如图 8-2 所示。一台汽车底盘从入口进入组装线,经过若干次零件装配,完成的汽车从组装线的末端出厂。每一条组装线有 n 道工序,编号为 $j=1,2,\cdots,n$。把第 i 号组装线(其中 i 为 1 或 2)上的第 j 道工序表为 $S_{i,j}$。1 号线上的第 j 道工序($S_{1,j}$)与 2 号线上的第 j 道工序($S_{2,j}$)做的工作是一样的。然而,由于工作台的建造时间以及技术不同,每道工序所需的时间不同,即使分处于两条组装线上编号一样的工序也是如此。用 $a_{i,j}$ 表示在工序 $S_{i,j}$ 所需的装配时间。如图 8-2 所示,底盘从入口进入一条组装线的 1 号工作台,然后从一道工序行进到下一道工序。底盘进入 i 号组装线(进入时间 e_i),完成的汽车从 i 号组装线出厂(出厂时间 x_i)。

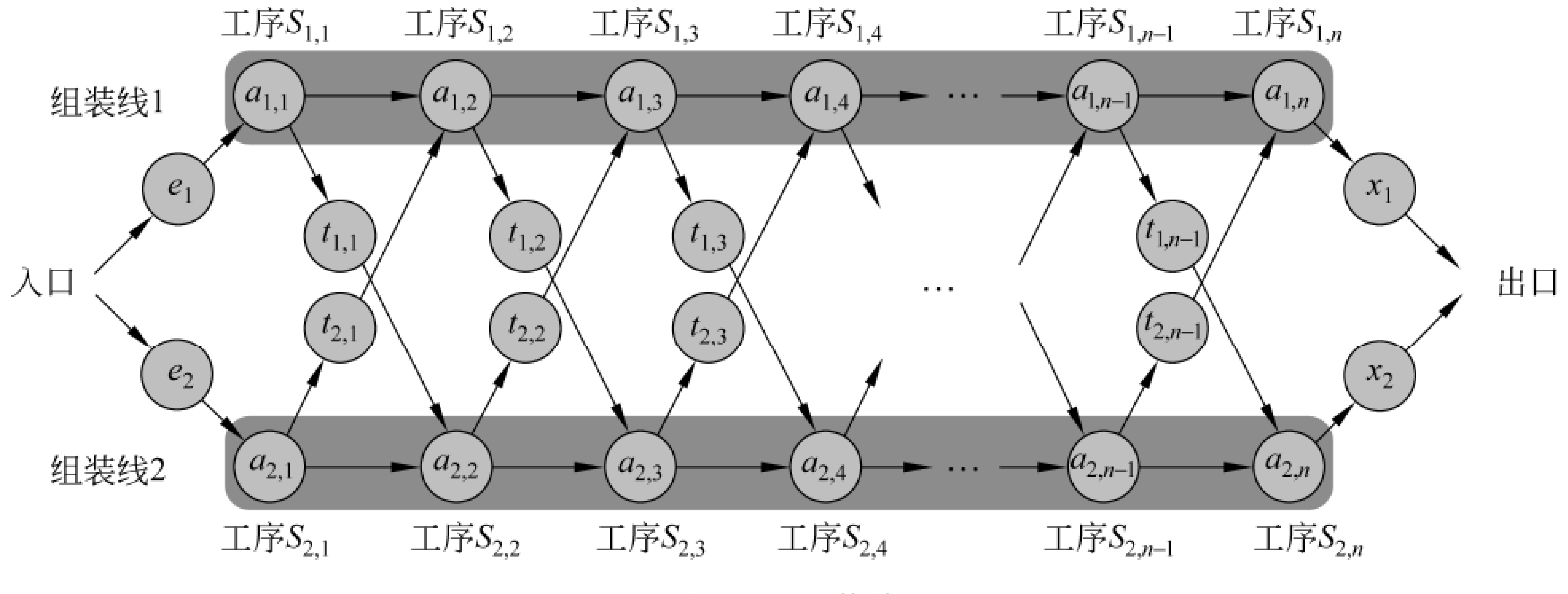
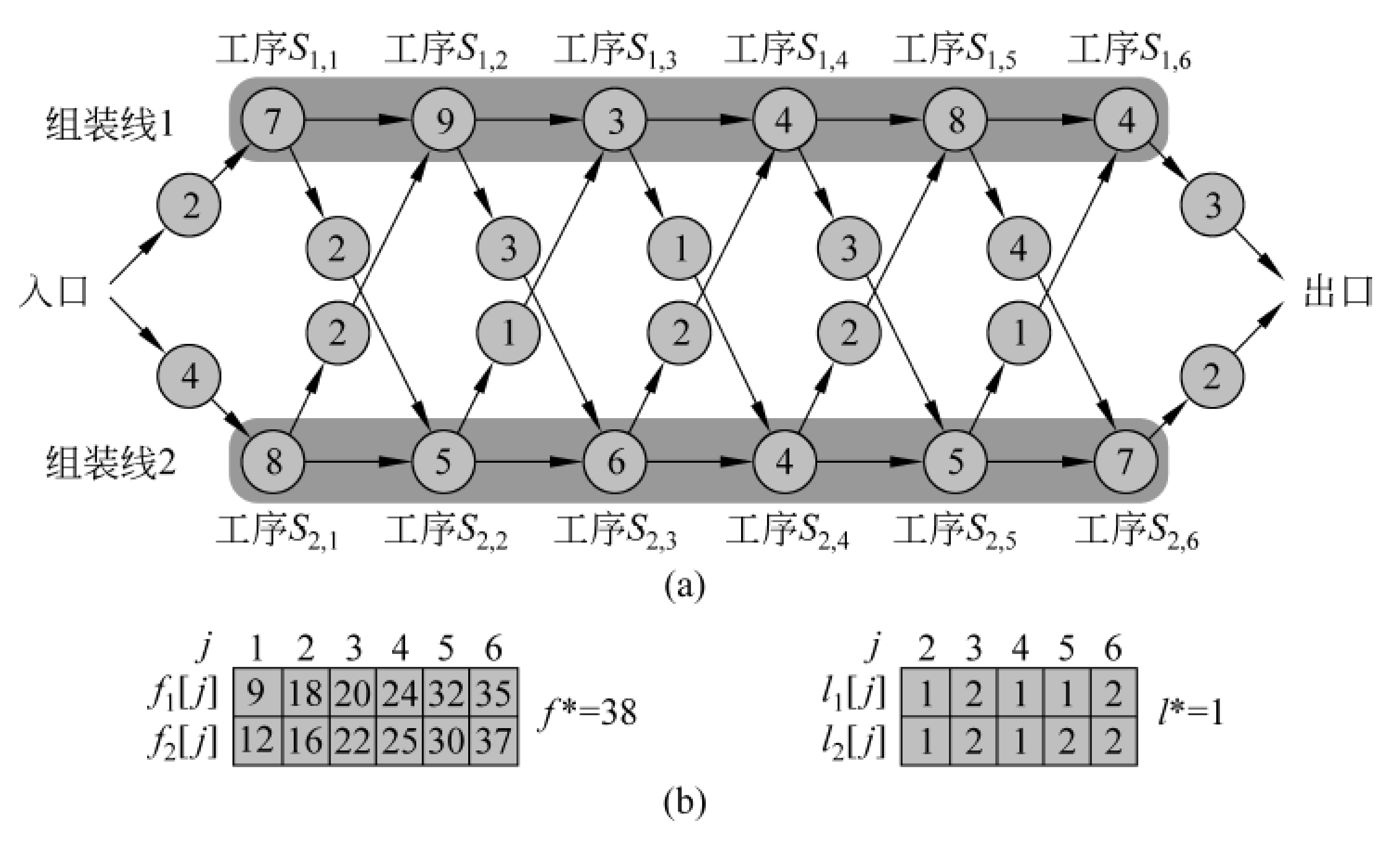


图 8-2 组装线

图 8-2 所示为找出工厂中最快路线的一个生产问题。有两条组装线，每一条有 n 道工序；i 号线上的第 j 号工序表为 $S_{i,j}$，在此工序所用的装配时间为 $a_{i,j}$。汽车底盘进入第 i 号组装线耗时 e_i，通过一条线上的第 j 号工序后，底盘可进入任一条线的第 $j+1$ 道工序。若继续在同一条线上则没有转移开销，但若从 $S_{i,j}$ 转移到另一条线上则要耗时 $t_{i,j}$。通过 n 工序装配完成的汽车从第 i 条线出厂耗时 x_i。问题是要确定从 1 号线上选择哪些工作台，从 2 号线上选择哪些工作台使得一辆汽车通过工厂的时间最少。

一台底盘从入口到出口必须通过所有 n 道工序才能完成组装。可以在一条线上完成 n 道工序，也可以在两条线上来回切换完成 n 道工序。在一条线上从一个工序到下一个工序，转移时间可忽略不计，而要在两条线之间转移则需要耗费转移时间。底盘完成 $S_{i,j}$ 工序后从一条线上转移到另一条线上的耗时为 $t_{i,j}$，其中 $i=1,2$ 以及 $j=1,2,\cdots,n-1$（因为 n 道工序后组装才完成）。问题是要确定从 1 号线上选择哪些工序，在 2 号线上选择哪些工序使得汽车通过工厂的时间最少。在图 8-3(a)的例子中，从 1 号线上选择工序 1、3、6，在 2 号线上选择 2、4、5 号工序所需时间最短。



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

图 8-3　组装线问题实例

一般地，组装线问题描述如下。

输入：两条组装线上的工序数 n，组装线上各道工序操作时间构成的矩阵 a，底盘在两条组装线间转移时间构成的矩阵 t，底盘从入口进入组装线所需时间构成的向量 e，完成组装后汽车下线出厂的时间构成的向量 x。

输出：由 n 道工序构成的序列 $\{S_{i_1,1},S_{i_2,2},\cdots,S_{i_n,n}\}$，$i_p\in\{1,2\}$，$p=1,2,\cdots,n$，使得底盘按此序列完成组装所需时间最少。

底盘从入口到组装完成出厂，序列 $S_{i_1,1},S_{i_2,2},\cdots,S_{i_n,n}$ 中的每一道工序 $S_{i_p,j}$ 可以是两条组装线中的任一条上的。根据乘法原理知 $S_{i_1,1},S_{i_2,2},\cdots,S_{i_n,n}$ 共有 2^n 个不同的情形。所以，本问题的解空间含有 2^n 个可能解，每个解对应确定的组装时间，我们希望算得所用时间最少的组装路线。这是一个组合优化问题，通过罗列所有可能的通过工厂的路径来确定最快的路径的“强力”算法将需要 $\Omega(2^n)$ 的时间，当 n 很大时这是不可行的。下面用动态规划的策略来解决此问题。

8.1.2 算法设计与分析

1. 最优子结构

动态规划策略的第一步是筹划最优解的结构。对于组装线问题,可以如下来完成这一步工作。考虑底盘从起点起通过 $S_{1,j}$ 的最优路径。若 $j=1$,已走过的路径是唯一的,所以很容易确定通过 $S_{1,j-1}$ 所用的时间。然而,对于 $j=2,3,\cdots,n$,有两种选择:从 $S_{1,j-1}$ 工序在同一条线直接到 $S_{1,j}$ 工序,从上一道工序 $j-1$ 转移到工序 j 的时间被忽略。另一个情况是,底盘可能来自 $S_{2,j-1}$ 且转移到 $S_{1,j}$ 的时间为 $t_{2,j-1}$。下面将分别考虑这两可能性,尽管它们有很多相同的地方。

首先,假定通过工序 $S_{1,j}$ 的最快路径经过工序 $S_{1,j-1}$。关键是要看到该路经中从起点开始到工序 $S_{1,j-1}$ 这一段是最快的,为什么?若有一条到 $S_{1,j-1}$ 的最快路径,可以把这条最快路径代换到通过 $S_{1,j}$ 的最快路径中将导致一条更快的通过 $S_{1,j}$ 的路径,这是一个矛盾。

类似地,假定通过 $S_{1,j}$ 的最快路径经过 $S_{2,j-1}$。现在注意到路径中底盘从起点开始到 $S_{2,j-1}$ 的一段是最快的。理由是一样的:如果还有一条从起点起到 $S_{2,j-1}$ 更快的路径,则将其替换到经过 $S_{1,j}$ 的最快路径,将导致一条经过 $S_{1,j}$ 的更快的路径,这将是一个矛盾。

利用最优结构,能够从子问题的最优解来构造原问题的最优解。对于组装线调度而言,考察通过 $S_{1,j}$ 的最快路径,它必通过 1 号线或 2 号线的第 $j-1$ 道工序。于是,通过的最快路径是如下两者之一。

(1) 此最快路径通过 $S_{1,j-1}$ 且直接通过 $S_{1,j}$。

(2) 此最快路径通过 $S_{2,j-1}$ 并从 2 号线转移到 1 号线,然后通过 $S_{1,j}$。

利用对称性推出,通过 $S_{2,j}$ 的最快路径为下列两者之一。

① 此最快路径通过 $S_{2,j-1}$ 且直接通过 $S_{2,j}$。

② 此最快路径通过 $S_{1,j-1}$ 并从 1 号线转移到 2 号线,然后通过 $S_{2,j}$。

为解通过两条线上第 j 道工序的最快路径,解通过两条线上第 $j-1$ 道工序的最快路径的子问题。

2. 最优解值的递归方程

接下来动态规划策略是用子问题最优解的值递归地定义最优解的值。对于组装线调度问题,把求通过两条线上第 $j(j=1,2,\cdots,n)$ 道工序的最快路径作为子问题。设 $f[i,j]$ 表示底盘从起点到通过工序 $S_{i,j}$ 的可能的最短时间。

我们的最终目标是确定底盘通过全厂的最短时间,用 f^* 来表示。底盘必须经过 1 号线或 2 号线上的 n 道工序,然后出厂。由于这两条线路的较快者为通过全厂的最快路径,所以有

$$f^* = \min(f[1,n]+x_1, f[2,n]+x_2) \tag{8-1}$$

推导 $f[1,1]$ 及 $f[2,1]$ 很容易。底盘是直接通过两条线的第 1 道工序的。于是

$$f[1,1] = e_1 + a_{1,1} \tag{8-2}$$

$$f[2,1] = e_2 + a_{2,1} \tag{8-3}$$

图 8-3(a)指定开销 e_i、$a_{i,j}$、$t_{i,j}$ 及 x_i 的组装线问题实例。粗线路径表示通过工厂的最快路径。图 8-3(b)部分为图 8-3(a)中实例的 $f_i[j]$、f^*、$l_i[j]$及 l^* 的值。

现在来考虑如何计算 $f[i,j]$，$j=2,3,\cdots,n$（并且 $i=1,2$）。着重于 $f[1,j]$，回忆通过工序 $S_{1,j}$ 的最快路径或者是经过 $S_{1,j-1}$ 工序然后直接到达工序 $S_{1,j}$，或经过 $S_{2,j-1}$，从 2 号线转移到 1 号线，然后通过 $S_{1,j}$。对第一种情形，有 $f[1,j]=f[j-1]+a_{1,j}$，而对于后者，$f[1,j]=f[2,j-1]+t_{2,j-1}+a_{1,j}$。于是，对 $j=2,3,\cdots,n$，

$$f[1,j]=\min(f[1,j-1]+a_{1,j},f[2,j-1]+t_{2,j-1}+a_{1,j}) \tag{8-4}$$

对称地，对 $j=2,3,\cdots,n$，

$$f[2,j]=\min(f[2,j-1]+a_{2,j},f[1,j-1]+t_{1,j-1}+a_{2,j}) \tag{8-5}$$

合并式(8-2)～式(8-5)得到递归式：

$$f[1,j]=\begin{cases}e_1+a_{1,1} & j=1\\ \min\,(f[1,j-1]+a_{1,j},f[2,j-1]+t_{2,j-1}+a_{1,j}) & j\geqslant 2\end{cases} \tag{8-6}$$

$$f[2,j]=\begin{cases}e_2+a_{2,1} & j=1\\ \min\,(f[2,j-1]+a_{2,j},f[1,j-1]+t_{1,j-1}+a_{2,j}) & j\geqslant 2\end{cases} \tag{8-7}$$

图 8-3(b)展示了利用式(8-6)及式(8-7)计算的图 8-3(a)部分中例子的 $f[i,j]$值和 f^* 的值。

值 $f[i,j]$给出了子问题最优解的值。为跟踪如何创建最优解，定义 $l[i,j]$为通过 $S_{i,j}$ 的最快路径经过第 $j-1$ 道工序时所在的组装线号，1 或 2。此处 $i=1,2$ 以及 $j=2,3,\cdots,n$（不定义 $l[i,1]$是因为工序 1 的前序不存在），还定义 l^* 为通过全厂的最快路径经过的第 n 道工序所在的组装线号。$l[i,j]$的值可以用来跟踪一条最快路径。利用图 8-3(b)中展示的 l^* 及 $l[i,j]$的值，可以如下追踪图 8-3(a)部分的一条最快路径。从 $l^*=1$ 开始，用到 $S_{1,6}$。考察$l[1,6]$，其值为 2，所以用到 $S_{2,5}$。继续考察 $l[2,5]=2$(用到 $S_{2,4}$)，$l[2,4]=1$(工序 $S_{1,3}$)，$l[1,3]=2$(工序 $S_{2,2}$)，以及 $l[2,2]=1$(工序 $S_{1,1}$)。

3. 计算最短时间

此时，可以用一个简单的方式写出一个基于式(8-1)及式(8-6)和式(8-7)来计算通过整个工厂的最快路径。但这样的一个递归算法存在一个问题：其运行时间是 $O(2^n)$。这是因为组装线问题具有子问题重叠性，即低层的子问题将多次被计算。为了解决这一点，设 $r_i(j)$为递归算法中对 $f[i,j]$的访问次数。根据式(8-1)，有：

$$r_1(n)=r_2(n)=1 \tag{8-8}$$

根据式(8-6)和式(8-7)，对 $j=1,2,\cdots,n-1$，有

$$r_1(j)=r_2(j)=r_1(j+1)+r_2(j+1) \tag{8-9}$$

下面说明对 $j=1,2,\cdots,n$，$r_i(j)=2^{n-j}$，也就是对 $j=0,1,\cdots,n-1$，

$$r_i(n-j)=2^j \tag{8-10}$$

为此，对 j 做数学归纳。当 $j=0$ 时，$r_i(n-j)=r_i(n)$，根据式(8-8)，$r_i(n)=1=2^0$，说明式(8-10)成立。假定对 $0<j<n-1$ 式(8-10)成立，即 $r_i(n-j)=2^j$。考虑 $j+1$ 的情形。

$$\begin{aligned}r_i(n-(j+1))&=r_i(n-j-1)\\&=r_1(n-j)+r_2(n-j) && \text{（根据式(8-9)）}\\&=2^j+2^j && \text{（根据归纳假设）}\\&=2^{j+1}\end{aligned}$$

这样就证明了式(8-10)的正确性,于是 $f[1,1]$就被访问了 2^{n-1}次。

注意,对 $j \geqslant 2$,每一个值 $f[i,j]$仅依赖于 $f[1,j-1]$和 $f[2,j-1]$的值。这样,可以自底向上地计算表中元素 $f[i,j]$的值,这样做比递归方法要好得多。按工序号 j 的递增——图 8-3(b)中从左到右的顺序可以在时间 $\Theta(n)$内计算出通过全厂的最快路径。

FASTEST-WAY(a,t,e,x,n)
1 $f[1,1] \leftarrow e_1 + a_{1,1}$
2 $f[2,1] \leftarrow e_2 + a_{2,1}$
3 **for** $j \leftarrow 2$ to n
4 　　**do if** $f[1,j-1] + a_{1,j} \leqslant f[2,j-1] + t_{2,j-1} + a_{1,j}$
5 　　　　**then** $f[1,j] \leftarrow f[1,j-1] + a_{1,j}$
6 　　　　　$l[1,j] \leftarrow 1$
7 　　　　**else** $f[1,j] \leftarrow f[2,j-1] + t_{2,j-1} + a_{1,j}$
8 　　　　　$l[1,j] \leftarrow 2$
9 　　**if** $f[2,j-1] + a_{2,j} \leqslant f[1,j-1] + t_{1,j-1} + a_{2,j}$
10 　　　　**then** $f[2,j] \leftarrow f[2,j-1] + a_{2,j}$
11 　　　　　$l[2,j] \leftarrow 2$
12 　　　　**else** $f[2,j] \leftarrow f[1,j-1] + t_{1,j-1} + a_{2,j}$
13 　　　　　$l[2,j] \leftarrow 1$
14 **if** $f[1,n] + x_1 \leqslant f[2,n] + x_2$
15 　　**then** $f^* = f[1,n] + x_1$
16 　　　　$l^* = 1$
17 　　**else** $f^* = f[2,n] + x_2$
18 　　　　$l^* = 2$

算法 8-1 计算最快路径组装时间的 FASTEST-WAY 过程

FASTEST-WAY 运行如下。

第 1 行和第 2 行利用式(8-2)和式(8-3)计算 $f[1,1]$和 $f[2,1]$。第 3～13 行的 **for** 循环对 $i=1,2$ 及 $j=2,3,\cdots,n$ 计算 $f[i,j]$和 $l[i,j]$。第 4～8 行利用式(8-4)计算 $f[1,j]$和 $l[1,j]$,第9～13 行利用式(8-5)计算 $f[2,j]$和 $l[2,j]$。最后,第 14～18 行利用式(8-1)计算 f^* 和 l^*。由于第 1 行和第 2 行及第 14～18 行消耗常数时间,第 3～13 行 **for** 循环的 $n-1$ 次重复的每一次消耗常数时间,整个过程的耗时为 $\Theta(n)$。

观察 $f[i,j]$和 $l[i,j]$的值的计算过程,方法之一是我们在表中填写的项。参考图 8-2(b),在含有值 $f[i,j]$及 $l[i,j]$的表中从左到右(每一列的数据是自顶向下的)填写。为填写项 $f[i,j]$,需要值 $f[1,j-1]$和 $f[2,j-1]$,而这两项已经计算出来且已存储,只要查表就得。

4. 构造通过整个工厂的最快路径

在计算了 $f[i,j]$、f^*、$l[i,j]$和 l^* 这些值以后,要来构造通过整个工厂的最快路径中所经过的各道工序的序列。利用记录在 l^* 中最优路径中汽车出厂前所做的最后一道工序所在的组装线编号,可以通过查询 $l[l^*,n]$追溯到最快路径中前一道工序所在组装线编号,这是因为 $l[i,j]$记录了最快路径中第 j 道工序的前一道工序所在的组装线编号。以这样的方

式，可以写出下列递归过程打印输出一个最优解。

```
PRINT-STATIONS(l,i,j)
if j=1
    then print "line " i ",station " j
        return
PRINT-STATIONS(l,l[l[i,j],j],j-1)
print "line " i ",station " j
```

算法 8-2　打印最快路径的 PRINT-STATIONS 过程

过程 PRINT-STATIONS 从最顶层调用时向参数 i 传递 l^*、向参数 j 传递 n 开始，每次递归，传递给参数 j 的值都要减少 1，故递归调用 $n-1$ 次。每次递归运行时间都仅消耗常数时间，所以过程 PRINT-STATIONS 的运行时间为 $\Theta(n)$。对图 8-3 中的例子，PRINT-STATIONS 将产生如下输出：

```
line 1,station 1
line 2,station 2
line 1,station 3
line 2,station 4
line 2,station 5
line 1,station 6
```

8.1.3　应用——牛牛玩牌

Cow Solitaire

Description

Late summer on the farm is a slow time，very slow. Betsy has little to do but play cow solitaire. For self-evident reasons，cow solitaire is not so challenging as any number of solitaire games played by humans.

Cow solitaire is played using an $N \times N(3 \leqslant N \leqslant 7)$ grid of ordinary playing cards with four suits (Clubs，Diamonds，Hearts，and Spaces) of 13 cards (Ace，2，3，4，…，10，Jack，Queen，King). Cards are named with two characters：their value (A，2，3，4，…，9，T，J，Q，K) followed by their suit (C，D，H，S). Below is a typical grid when $N=4$：

8S AD 3C AC　(Eight of Spades，Ace of Diamonds，etc.)

8C 4H QD QS

5D 9H KC 7H

TC QC AS 2D

To play this solitaire game，Betsy starts in the lower left corner (TC) and proceeds using exactly $2*N-2$ moves of 'right' or 'up' to the upper right corner. Along the way，she accumulates points for each card (Ace is worth 1 point，2 is worth 2 points，…，9 is worth 9 points，T is worth 10 points，J is 11，Q is 12，and K is 13) she traverses. Her

goal is to amass the highest score.

If Betsy's path was TC-QC-AS-2C-7H-QS-AC, her score would be 10+12+1+2+7+12+1=45. Had she taken the left side then top (TC-5D-8C-8S-AD-3C-AC), her score would be 10+5+8+8+1+3+1=36, not as good as the other route. The best score for this grid is 69 points (TC-QC-9H-KC-QD-QS-AC=>10+12+9+13+12+12+1). Betsy wants to know the best score she can achieve. One of the geek cows once told her something about "working from the end back to the beginning," but she didn't understand what they meant.

Input

Line 1: A single integer: N.

Lines 2..N+1: Line $i+1$ lists the cards on row i (row 1 is the top row) using N space-separated card names arranged in the obvious order.

Output

Line 1: A single line with an integer that is the best possible score Betsy can achieve.

Sample Input

```
4
8S AD 3C AC
8C 4H QD QS
5D 9H KC 7H
TC QC AS 2D
```

Sample Output

```
69
```

1. 问题描述与分析

用 $N\times N$ 张扑克牌排成一个方阵。每张牌的点数形成一个 $N\times N$ 的整数网格 A,从左下角 $A[N,1]$漫游(每一步只能向上或向右行走一步)到右上角 $A[1,N]$,要求漫游所经路径(每条路径都要经过 $2N-1$ 个格子)上各个格子中整数之和最大。这是一个组合优化问题。从网格的左下角 $A[N,1]$漫游到右上角 $A[1,N]$有很多路径,每条路径上的 $2N-1$ 个数之和为该条路径的目标值,目的是找到一条目标值最大的路径的目标值。问题形式化描述如下。

输入:$N\times N$ 整数网格 $A[1..N,1..N]$。

输出:A 中从左下角 $A[N,1]$到右上角 $A[1,N]$最优路径(目标值最大)的目标值。

由于每个可能解中除两端格子固定外,其余格子均可有两种选择:向上或向右。所以,解空间含有 2^{2N-3} 个元素。显然当 N 很大时,对解决此组合优化问题的强力算法不是好的选择。

下面来说明该问题具有最优子结构。假定路径 p 是从 A 的左下角 $A[N,1]$到右上角 $A[1,N]$的一条最优路径,$A[i,j]$是路径 p 中的一个格子。则 p 中从 $A[N,1]$到 $A[i,j]$的部分 p_1 恰为从 $A[N,1]$到 $A[i,j]$的一条最优路径,同时 p 中从 $A[i,j]$ 到 $A[1,N]$ 的部分 p_2 恰为从 $A[i,j]$ 到 $A[1,N]$ 的一条最优路径。这是因为如果从 $A[N,1]$到 $A[i,j]$有一条比 p_1 更优的路径 p_1',则 p_1'与 p_2 将合并成一条从 $A[N,1]$到 $A[1,N]$的一条比 p 更好的路

径，这与 p 是最优的路径假设矛盾。同理可知对 p_2 的结论。按此性质，若设 $f[i,j]$ 为从 $A[N,1]$ 到 $A[i,j]$ 的最优路径目标值，则：

$$f[i,j]=\begin{cases}0 & i=N+1 \text{ 或 } j=0 \\ \max\{f[i+1,j],f[i,j-1]\}+A[i,j] & 0<i\leqslant N,0<j\leqslant N\end{cases} \tag{8-11}$$

我们要求的是 $f[1,N]$。从图 8-4 中就可知问题具有重叠子问题性质。于是用动态规划策略设计解决此问题的算法。

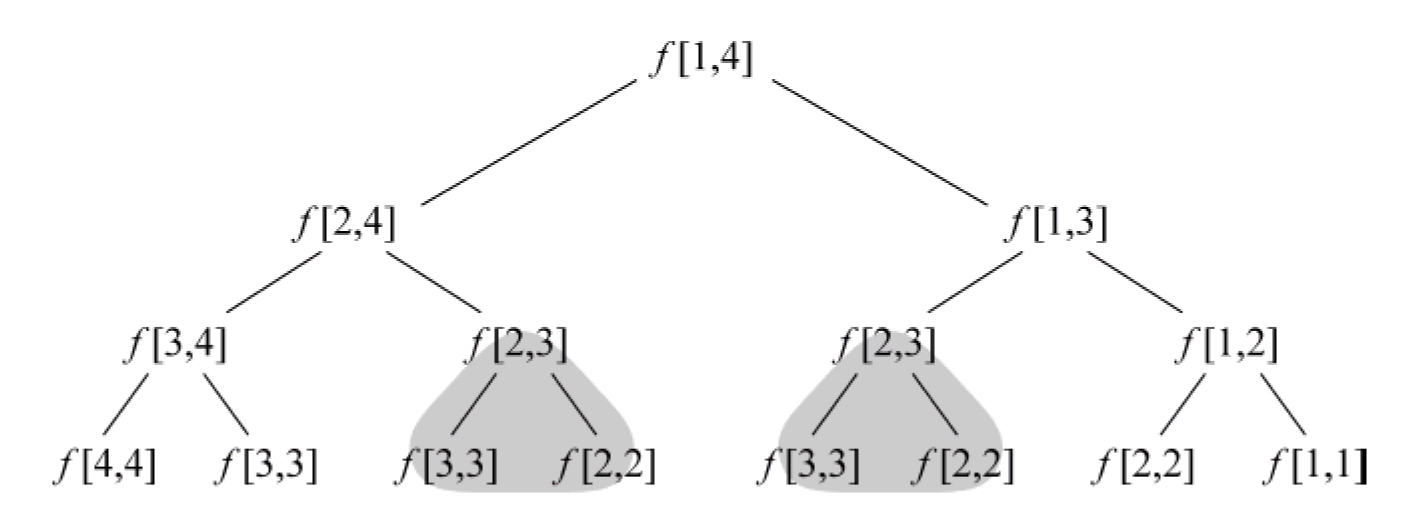


图 8-4　Cow Solitaire 问题的重叠子问题性质

2. 算法描述

根据问题的最优子结构及子问题重叠性，可以写出下列自底向上及表计算的动态规划算法。

```
COW-SOLITAIRE(A)                          ▷二维数组 A[1..N,1..N]表示牌点网格
1 N←rows[A]
2 for i←0 to N+1
3       do f[i,0]←0
4 for j←1 to N
5       do f[N+1,j]←0
6 for i←N downto 1
7       do for j←1 to N
8              do q←max{f[i+1,j],f[i,j-1]}
9                 f[i,j]←q+A[i,j]
10 return f[1,N]
```

算法 8-3　解决 Cow Solitaire 问题的 COW-SOLITAIRE 过程

算法 COW-SOLITAIRE 运行如下。

第 2 行和第 3 行及第 4 行和第 5 行的 **for** 循环按式(8-11)计算最低层子问题最优解的值 $f[i,0](i=1,2,\cdots,N+1)$ 及 $f[N+1,j](j=0,1,\cdots,N)$。第 6～9 行的 **for** 循环按式(8-11)，自底向上地逐层计算 $f[i,j]$。显然，此算法的时间复杂度为 $\Theta(N^2)$。

3. 程序实现

1）计算最优解

在 C 语言中实现算法 8-3 是很直接的。

```
1 int cowSolitaire(int *a,int n){
2     int *f=(int *)malloc((n+2)*(n+1)*sizeof(int)),    /*为数表 f 分配空间*/
3         i,j,q;
4     assert(f);
5     for(i=0;i<=n+1;i++)
6         f[i*(n+1)]=0;                                 /*f[i,0]←0*/
7     for(j=1;j<n+1;j++)
8         f[(n+1)*(n+1)+j]=0;                           /*f[n+1,j]←0*/
9     for(i=n;i>0;i--)
10        for(j=1;j<=n;j++){
11            q=f[(i+1)*(n+1)+j]>=f[i*(n+1)+j-1]?
                                                        /*q←max{f[i+1,j],f[i,j-1]}*/
12                f[(i+1)*(n+1)+j]:
13                f[i*(n+1)+j-1];
14            f[i*(n+1)+j]=q+a[(i-1)*n+j-1];            /*f[i,j]←q+a[i,j]*/
15        }
16    q=f[n+1+n];                                       /*f[1,n]作为返回值*/
17    free(f);
18    return q;
19 }
```

程序 8-1　实现算法 8-3 的 C 函数

对程序 8-1 的说明如下。

(1) 函数 cowSolitaire 有 2 个参数,表示牌点(整数)网格的数组 a(它对应算法 8-3 的过程的参数 A)和 a 的阶数 n。注意,为了提高程序的运行效率,我们用一维数组,按行优先原则存储二维数组的数据。函数返回从网格 a 的左下角到右上角最佳路径的牌点总和(整数)。

(2) 和算法过程一样,函数内定义了用来记录各层子问题最优解值的数表 f,第 2 行将其定义为整型指针,并为其分配存储空间。注意,算法中数表 f 的结构是一个$(N+1)\times(N+1)$的二维数组 $f[1..N+1,0..N]$。为使程序代码更接近于算法伪代码,为 f 分配$(n+2)\times(n+1)$的空间,舍弃 f 的下标为 0 的那一行。此外,局部变量 i、j、q 的意义与算法过程的同名变量是一致的。

(3) 第 5 行和第 6 行与第 7 行和第 8 行的 **for** 循环分别对应算法中第 2 行和第 3 行及第 4 行和第 5 行的 **for** 循环。注意,按行优先原则存储在一维数组中的二维数组的数据元素。于是,第 6 行中 f[i*(n+1)]表示的是 f[i,0],而第 8 行中的 f[(n+1)*(n+1)+j]表示的是 f[n+1,j]。第 9~15 行的两重嵌套 **for** 循环对应算法中第 6~9 行的两重嵌套 **for** 循环。其中第 11~13 行用一个条件表达式计算 max{f[i+1,j],f[i,j-1]}。由于 f 是指向动态数组的指针,为防止内存泄漏,函数退出前,第 17 行将其释放掉。

2) **主函数**

调用程序 8-1 中的函数 cowSolitaire,解决 Cow Solitaire 问题的主函数代码如下。

```
int main(){
    FILE *f1=fopen("chap04/Cow Solitaire/inputdata.txt","r"),
```

```
3         *f2=fopen("chap04/Cow Solitaire/outputdata.txt","w");
4     char s[4];
5     int *a,n,i,j;
6     fscanf(f1,"%d",&n);
7     a=(int*)malloc(n*n*sizeof(int));
8     for(i=0;i<n;i++)
9         for(j=0;j<n;j++){
10            fscanf(f1,"%s",s);
11            a[i*n+j]=getValue(s[0]);
12        }
13    fprintf(f2,"%d\n",cowSolitaire(a,n));
14    free(a);
15    fclose(f1); fclose(f2);
16    return 0;
17 }
```

程序 8-2　解决 Cow Solitaire 问题的 C 程序

程序中第 8 行和第 9 行的 **for** 循环从输入文件 f1 中读取牌局数据，填写二维数组 a(存储在一维数组中)，第 13 行调用函数 cowSolitaire 计算对牌局数据 a 计算的最优解的值写入输出文件 f2。

第 10 行从输入文件中读取牌局中的一项数据 s，s 是一个含有 2 个字符的字符串，s[0]表示牌点数，s[1]表示牌的花色。填写到数组 a 的是 s[0]表示的数据，而它表示的是'2'～'9','A','J','Q','K'的字符，所以第 11 行调用函数 getValue 将这样的字符数据转换成对应的数值数据。该函数的实现代码如下。

```
int getValue(char x){                /*将 x 中记录的牌点符号转换成整数值*/
    if(x>='2'&&x<='9')
        return x-'0';
    if(x=='A')
        return 1;
    if(x=='T')
        return 10;
    if(x=='J')
        return 11;
    if(x=='Q')
        return 12;
    return 13;
}
```

程序 8-3　将字符转换为整数值的 C 函数

将程序 8-1～程序 8-3 存在文件夹 chap08/Cow Solitaire 的源文件 Cow Solitaire.c 中，读者可打开研读，并试运行。

8.2 最长公共子序列

8.2.1 问题描述

在很多应用中需要比较两个序列的“相似性”,例如,两个 DNA 序列的比对。人们常用公共子序列的长度来描述这种相似性。

已知序列的**子序列**是在已知序列中去掉零个或多个元素后形成的序列。例如,$Z=<B,C,D,B>$是 $X=<A,B,C,B,D,A,B>$的一个子序列。

给定两个序列 X 和 Y,若 Z 同时为 X 和 Y 的子序列,称序列 Z 是 X 和 Y 的一个公共子序列。X 和 Y 的公共子序列中长度最大者,称为 X 和 Y 的**最长公共子序列**(Common Longest Subsequence,LCS)。例如,若 $X=<A,B,C,B,D,A,B>$ 且 $Y=<B,D,C,A,B,A>$,序列$<B,C,A>$是 X 和 Y 的一个公共子序列。然而,它不是 X 和 Y 的一个 LCS,这是因为它的长度为 3,而长度为 4 的序列$<B,C,B,A>$也是 X 和 Y 的一个公共子序列。序列$<B,C,B,A>$是 X 和 Y 的一个 LCS,$<B,D,A,B>$也是,这是因为没有长度为 5 或更大的公共子序列了。将此问题形式化为如下形式。

输入:序列 $X=<x_1,x_2,\cdots,x_m>$和 $Y=<y_1,y_2,\cdots,y_n>$。

输出:X 与 Y 的一个最长公共子序列 Z。

这是一个组合优化问题。X 与 Y 的每个公共子序列(可行解)都有一个长度(目标值),要求计算最长公共子序列(最优解)。如果用强力算法来解决这个问题,则需要在 X、Y 的所有子序列中找出公共子序列,然后从中选择长度最大的。这需要指数级的时间来完成,因为一个长度为 n 的序列,有 2^n 个不同的子序列。

8.2.2 算法设计与分析

1. 最优子结构与子问题重叠性

为了考察这个问题能否用动态规划的方法加以解决,需要验证该问题是否具有最优子结构和子问题重叠的特性。为此,定义序列 X 的第 i 个**前缀**为 $X_i=<x_1,x_2,\cdots,x_i>$,$i=0,1,\cdots,m$。例如,若 $X=<A,B,C,B,D,A,B>$,则 $X_4=<A,B,C,B>$,而 X_0 是空序列。用以下定理说明最长公共子序列问题具有最优子结构特征。

定理 8-1(最长公共子序列的最优子结构) 设 $X=<x_1,x_2,\cdots,x_m>$和 $Y=<y_1,y_2,\cdots,y_n>$为两个序列,并设 $Z=<z_1,z_2,\cdots,z_k>$为 X 和 Y 的任一 LCS。

(1) 若 $x_m=y_n$,则 $z_k=x_m=y_n$ 且 Z_{k-1}是 X_{m-1}和 Y_{n-1}的一个 LCS。

(2) 若 $x_m\neq y_n$,且 $z_k\neq x_m$,则 Z 是 X_{m-1}和 Y 的一个 LCS。

(3) 若 $x_m\neq y_n$,且 $z_k\neq y_n$,则 Z 是 X 和 Y_{n-1}的一个 LCS。

证明 (1) 若 $z_k\neq x_m$,则可以把 $x_m=y_n$ 追加到 Z 而得到一个长度为 $k+1$ 的 X 和 Y 的公共子序列,这与 Z 是 X 和 Y 的最长公共子序列的假设矛盾。于是 $z_k=x_m=y_n$。显然,

Z_{k-1}是 X_{m-1}和 Y_{n-1}的一个长度为 $k-1$ 的公共子序列。我们希望能说明它还是一个 LCS。用反证法，假定 X_{m-1} 和 Y_{n-1} 有一个长度比 $k-1$ 更大的公共子序列。将 $x_m=y_n$ 追加到 W 产生出 X 和 Y 的一个公共子序列，其长度大于 k，这样就矛盾。

(2) 若 $z_k \neq x_m$，则 Z 是 X_{m-1}和 Y 的一个公共子序列。若 X_{m-1}和 Y 的一个公共子序列 W 的长度比 k 还大，则 W 还是 X_m 和 Y 的一个公共子序列，这与 Z 是 X 和 Y 的 LCS 矛盾。

(3) 的证明和(2)的证明是对称的。

定理 8-1 说明两个序列的 LCS 包含了这两个序列的前缀的 LCS。于是 LCS 问题具有最优子结构特性。根据此定理，若 $x_m=y_n$，必须寻求 X_{m-1}和 Y_{n-1}的一个 LCS。把 $x_m=y_n$ 追加到此 LCS 的尾部就得到 X 和 Y 的一个 LCS。若 $x_m \neq y_n$，则必须解两个子问题：寻求 X_{m-1}和 Y 的一个 LCS 及 X 和 Y_{n-1}的一个 LCS，两者中的较长者就是 X 和 Y 的一个 LCS。由于这些情况穷尽了所有的可能，我们知道这些子问题最优解之一必包含在 X 和 Y 的一个 LCS 中。

设 $c[i,j]$为子序列 X_i 和 Y_j 的 LCS 的长度。若 $i=0$ 或 $j=0$，这两个子序列中至少有一个的长度为 0，所以 LCS 的长度为 0。最长公共子序列问题的最优子结构给出下列递归式：

$$c[i,j]=\begin{cases}0 & i=0 \text{ 或 } j=0 \\ c[i-1,j-1]+1 & i,j>0 \text{ 且 } x_i=y_j \\ \max\{c[i,j-1],c[i-1,j]\} & i,j>0 \text{ 且 } x_i \neq y_j\end{cases} \tag{8-12}$$

在式(8-12)中不难看到重叠子问题特性：为寻求 X 和 Y 的最长公共子序列，要寻求 X 和 Y_{n-1}以及 X_{m-1}和 Y 的最长公共子序列。而这些子问题有着寻求 X_{m-1}和 Y_{n-1}的最长公共子序列子子问题。很多其他子问题共享子子问题，如图 8-5 所示。于是，利用动态规划策略，自底向上地计算 $c[i,j]$，直至算出 $c[m,n]$。注意，在此定义中，二维表 c 的行标和列标都是从 0 开始编号的。

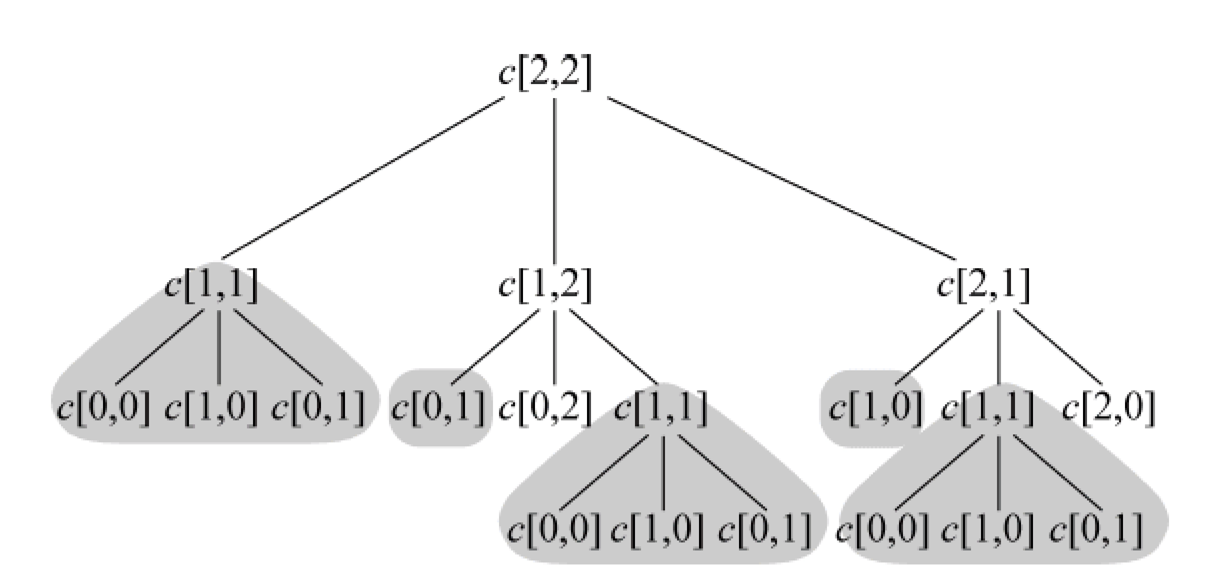


图 8-5　LCS 问题的子问题重叠性

2. 计算 LCS 长度

```
LCS-LENGTH (X,Y)
m←length[X]
n←length[Y]
for i←1 to m
    do c[i,0]←0
for j←0 to n
```

```
6        do c[0,j]←0
7 for i←1 to m
8        do for j←1 to n
9               do if x_i = y_j
10                     then c[i,j]←c[i-1,j-1]+1
11                     else if c[i-1,j]≥c[i,j-1]
12                             then c[i,j]←c[i-1,j]
13                             else c[i,j]←c[i,j-1]
15 return c
```

算法 8-4 计算两个序列的最长公共子序列的 LCS-LENGTH 算法

算法过程中第 3 行和第 4 行的 **for** 循环及第 5 行和第 6 行的 **for** 循环按式(8-12)计算最底层的子问题最优解的值。第 7～13 行的两重嵌套 **for** 循环按式(8-12)自底向上地计算各层子问题最优解的值。例如,对 $X=<A,B,C,B,D,A,B>$ 和 $Y=<B,D,C,A,B,A>$ 算法产生的表 c 如下。

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_j	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

3. 构造一个最优解

可以用如下过程来根据过程 LCS-LENGTH 计算出来的表格 c 构造出最优解。

```
PRINT-LCS(c,X,Y,i,j)
if i=0 or j=0
       then return
if x_i = y_j
       then PRINT-LCS (c,X,Y,i-1,j-1)
              print x_i
       elseif c[i-1,j]≥c[i,j-1]
              then PRINT-LCS (c,X,Y,i-1,j)
              else PRINT-LCS (c,X,Y,i,j-1)
```

算法 8-5 打印最长公共子序列的算法

4. 算法的运行时间

过程 LCS-LENGTH 的主体是第 7～16 行的两重嵌套的 **for** 循环，容易看出其时间复杂度为 $T(m,n)=\Theta(mn)$，其中 m、n 分别为 X 和 Y 的长度。过程 PRINT-LCS 的时间复杂度为 $T(m,n)=\Theta(m+n)$。

8.2.3　程序实现

我们的目标是开发一个能用来计算任何类型的两个序列的最长公共子序列的函数。为此，对算法 8-4 和算法 8-5 分别加以讨论。

1. 计算 LCS 的长度

如上所述，我们的目标是开发一个能计算两个任意类型的序列 x、y 的最长公共子序列的通用函数。如在前两章中看到的，C 语言中数据抽象的有力工具就是 **void** * 指针和函数指针。

```
1 int* lcsLength(void *x,void *y,int size,int m,int n,int(*comp)(void *,void *)){
2     int i,j;
3     int *c=(int *)malloc((m+1)*(n+1)*sizeof(int));
4     for(i=1;i<=m;i++)
5         c[i*(n+1)]=0;                                  /*c[i,0]←0*/
6     for(j=0;j<=n;j++)
7         c[j]=0;                                        /*c[0,j]←0*/
8     for(i=1;i<=m;i++)
9         for(j=1;j<=n;j++)
10            if(comp(x+(i-1)*size,y+(j-1)*size)==0)     /*if x[i]=y[j]*/
11                c[i*(n+1)+j]=c[(i-1)*(n+1)+j-1]+1;     /*c[i,j]←c[i-1,j-1]+1*/
12            else if(c[(i-1)*(n+1)+j]>=c[i*(n+1)+j-1])
13                c[i*(n+1)+j]=c[(i-1)*(n+1)+j];         /*c[i,j]←c[i-1,j]*/
14            else
15                c[i*(n+1)+j]=c[i*(n+1)+j-1];           /*c[i,j]←c[i,j-1]*/
16    return c;
17 }
```

程序 8-4　实现算法 8-4 的 C 函数

对程序 8-4 的说明如下。

(1) 函数 lcsLength 除了序列 x 和 y 以外，还多出了 size、m、n 和 comp 4 个参数。其中，m 和 n 分别是 x 和 y 所含的元素个数。size 是存储在 x、y 中的元素的真实长度。comp 是检测 x、y 中元素是否相等的比较规则。该函数返回计算所得的矩阵 c(用一维数组表示)。

(2) 用一维数组来表示矩阵。第 3 行将数组 c 定义成能表示(n+1)×(n+1)矩阵。注意，在此算法的实现中，数组 c 的下标编号与伪代码中是一致的，从 0 开始。

(3) 第 4～16 行实现的是算法 LCS-LENGTH 的第 3～15 行的操作,代码几乎是一一对应的。要注意的是用一维数组表示矩阵时,矩阵元素的行标、列标与数组下标的对应关系(见各行注释信息),以及第 10 行调用 comp 检测 x[i]、y[j]是否相等。

2. 构造一个最优解

算法 8-5 的实现如下。

```
1 void printLcs(int *c,int n,void *x,void *y,int size,int i,int j,
2                    int(*comp)(void *,void *),void(*prt)(void *)){
3    if(i==0 || j==0)
4        return;
5    if(comp(x+(i-1)*size,y+(j-1)*size)==0){              /* if x[i]=y[j] */
6        printLcs(c,n,x,y,size,i-1,j-1,comp,prt);
7        prt(x+(i-1)*size);                                /* print x[i] */
8    }else if(c[(i-1)*(n+1)+j]>=c[i*(n+1)+j-1])
9        printLcs(c,n,x,y,size,i-1,j,comp,prt);
10   else
11       printLcs(c,n,x,y,size,i,j-1,comp,prt);
12 }
```

程序 8-5 实现算法 8-5 的 C 函数

出于函数通用性的考虑,该函数除了对应于算法的表示矩阵 c,两个序列 x、y,以及两个下标的 i、j 的参数以外,还带有反映矩阵 c 的列数 n,表示序列中元素存储长度的 size,序列元素比较规则的 comp 和打印序列元素的操作 prt。

代码几乎是一一对应的。注意,第 5 行调用 comp 检测 x[i]、y[j]是否相等,第 7 行调用 prt 打印 x[i]。

为便于重用,将程序 8-4 和程序 8-5 定义的函数写在文件夹 dprog 中的源文件 lcs. c 中,而将这两个函数的原型声明写在同一文件夹内的头文件 lcs. h 中。

8.2.4 应用

1. 射雕英雄

Hero Shoot Eagle

Description

Once a hero named Guo Jing invented a new style of cross-bow that could shoot consecutively. The arrow could hit the eagle exactly if only the arrow could reach the height of the eagle. However,there was a flaw of the corss-bow: only the first arrow could reach any height and the height that the arrow shot latter could reach was always lower than the former one. Actually,the higher the cross-bow can hit,the better the performance it is. One day,Guo Jing happened to see a crowd of eagles flying through the sky. Now

you have to work out a program that helps Guo Jing to count how many eagles he can shoot down at most.

Input

This problem contains multiple test cases. For each test case, input the number of eagles n ($1 \leqslant n \leqslant 1000$) in the first line, then input the height h ($1 \leqslant h \leqslant 10000$) of each eagle in another new line with a blank between every two of them.

Output

Please calculate the maximal number of eagles m that can be shot down by the crossbow in the first line, and then output the height of the each eagle that have been shot down and separated by blanks in the second line.

Sample Input

```
8
389 207 155 300 299 170 158 65
2
100 105
```

Sample Output

```
6
389 300 299 170 158 65
1
105
```

1）问题描述与分析

郭靖发明了一种连发的弓箭。每支箭的射出高度可以事先确定，但都不会超过它的前面那支箭的射出高度。对飞过的鹰群（每只鹰的飞翔高度已知），最多能用此弓箭射下多少只鹰？本问题实质上是下列最长有序子序列问题。设 $A=<a_1,a_2,\cdots,a_n>$ 是 n 个不同的实数的序列，L 的有序子序列是这样一个子序列 $A'=<a_{k1},a_{k2},\cdots,a_{km}>$，其中 $k_1<k_2<\cdots<k_m$ 且 $a_{k1} \leqslant a_{k2} \leqslant \cdots \leqslant a_{km}$（或 $a_{k1} \geqslant a_{k2} \geqslant \cdots \geqslant a_{km}$）。而最长有序子序列问题是求最大的 m 值。本问题实际上是计算序列的最长下降子序列。

输入：序列 $A=<a_1,a_2,\cdots,a_n>$。

输出：A 的下降子序列 $A'=<a_{k1},a_{k2},\cdots,a_{km}>$，其中 $1 \leqslant k_1<k_2<\cdots<k_m \leqslant n$ 且 $a_{k1} \geqslant a_{k2} \geqslant \cdots \geqslant a_{km}$ 中的最长者。

要解决这个问题，构造一个新的序列 B，它是对 A 进行降序排序而得，可将此问题转换成求 A、B 的最长公共子序列。利用 LCS-LENGTH 和 PRINT-LCS 过程就可得到本问题的解。

2）算法描述

```
HERO-SHOOT-EAGLE(A)
n←legth[A]
copy A to B
SORT(B)                          ▷对 B 做降序排序
c←LCS-LENGTH(A,B)
```

```
print c[n,n] as a line
PRINT-LCS (c,A,n,n)
```

算法 8-6 解决 Hero Shoot Eagle 问题的伪代码过程

3) 程序实现

利用程序 8-2 和程序 8-3,很容易将算法 8-6 实现为以下程序。

```
1 int heroShootEagle(int *a,int n){
2     int *b=(int*)malloc(n*sizeof(int)),*c,x;
3     assert(b);
4     memcpy(b,a,n*sizeof(int));
5     quickSort(b,sizeof(int),0,n-1,intLess);
6     c=lcsLength(b,a,sizeof(n),n,n,intLess);
7     x=c[n*(n+1)+n];
8     printLcs(c,n,b,a,sizeof(int),n,n,intLess,putInt);
9     free(b);free(c);
10    return x;
11 }
12 int main(){
13    FILE *f1=fopen("chap08/Hero Shoot Eagle/inputdata.txt","r"),
14         *f2=fopen("chap08/Hero Shoot Eagle/outputdata.txt","w");
15    int n,*a,i;
16    initStrOutputStream(&ssout,500);              /*初始化全局串输出流 ssout*/
17    while(!feof(f1)){
18        fscanf(f1,"%d",&n);                       /*读入案例所含元素数目*/
19        a=(int*)malloc(n*sizeof(int));
20        for(i=0;i<n;i++)                          /*读入案例的 n 个数据*/
21            fscanf(f1,"%d",a+i);
22        fprintf(f2,"%d\n",heroShootEagle(a,n);    /*处理数据,并输出最长度*/
23        free(a);
24        fputs(ssout.begin,f2);                    /*输出最长降序子序列*/
25        fputc('\n',f2);
26        sosRewind(&ssout);                        /*串输出流复位*/
27    }
28    fclose(f1);fclose(f2);
29    return 0;
30 }
```

程序 8-6 解决 Hero Shoot Eagle 问题的 C 程序

对程序 8-6 的说明如下。

(1) 第 1～11 行定义的函数 heroShootEagle 实现算法 8-6。与算法相比,该函数除了数组参数 a 以外,还有一个表示数组 a 的元素个数的参数 n。函数需要向外界返回 a 中最长降序子序列的长度。

(2) 在 heroShootEagle 的函数体中,为数组 b 分配了存储空间后(第 2 行),调用库函数

memcpy 将数组 a 复制给数组 b(第 4 行)。第 5 行调用在第 3 章 3.2.2 节定义的函数 quickSort 对 b 进行降序排序。第 6 行调用本节中定义的函数 lcsLength 计算 b 与 a 的最长公共子序列长度,返回得到数表 c。第 7 行将 b 和 a 的最长公共子序列的长度 c[n,n]暂存于 x。第 8 行调用本节定义的函数 printLcs,将 b 和 a 的最长公共子序列输出到第 2 章中 2.1.4 节定义的全局串输出流 ssout 中(注意,传递给 printLcs 的最后一个参数是在第 2 章中 2.1.4 节定义的函数 putInt,该函数将传递给它的整型数据写入 ssout)。最后,第 10 行将存于 x 中的 b 与 a 的最长公共子序列长度作为函数值返回。

(3) 函数 heroShootEagle 与算法过程 HERO-SHOOT-EAGLE 有一些微妙的差别。算法过程将直接向输出文件输出最长降序子序列长度和最长降序子序列。而函数却是将最长降序子序列长度作为函数值返回,而将最长降序子序列写入到串输出流 ssout。这样做主要是因为考虑重用 lcsLength 和 printLcs。因为 printLcs 可以通过函数指针参数 proc 来灵活地处理各种类型的序列元素。我们的目标是将序列的元素写到指定的文件中去,但 proc 仅具有一个指向元素数据的指针作为参数,无法得到目标文件的信息。解决方法无非是定义新的向全局文件写入整型数据的函数或者利用已经定义好的全局串输出流 ssout 和向 ssout 写入整型数据函数 putInt。在此,选择后者,这既可以节省版面,又体现了代码重用的软件工程思想。这种做法,使得将向输出文件写入数据的操作移到了函数 heroShootEagle 之外的 main 函数中完成。第 22 行,调用 fprintf 函数将 heroShootEagle 返回值表示的最长降序子序列长度写入文件 f2,第 26 行调用函数 fputs 将记录在 ssout 中的最长降序子序列写入文件 f2。

程序 8-6 存储在文件夹 chap08/Hero Shoot Eagle 中的源文件 HeroShootEagle.c 中。

2. 基因函数

Human Gene Functions

Description

It is well known that a human gene can be considered as a sequence, consisting of four nucleotides, which are simply denoted by four letters, A, C, G, and T. Biologists have been interested in identifying human genes and determining their functions, because these can be used to diagnose human diseases and to design new drugs for them.

A human gene can be identified through a series of time-consuming biological experiments, often with the help of computer programs. Once a sequence of a gene is obtained, the next job is to determine its function.

One of the methods for biologists to use in determining the function of a new gene sequence that they have just identified is to search a database with the new gene as a query. The database to be searched stores many gene sequences and their functions and many researchers have been submitting their genes and functions to the database and the database is freely accessible through the Internet.

A database search will return a list of gene sequences from the database that are similar to the query gene.

Biologists assume that sequence similarity often implies functional similarity. So, the function of the new gene might be one of the functions that the genes from the list have. To exactly determine which one is the right one another series of biological experiments will be needed.

Your job is to make a program that compares two genes and determines their similarity as explained below. Your program may be used as a part of the database search if you can provide an efficient one.

Given two genes AGTGATG and GTTAG, how similar are they? One of the methods to measure the similarity of two genes is called alignment. In an alignment, spaces are inserted, if necessary, in appropriate positions of the genes to make them equally long and score the resulting genes according to a scoring matrix.

For example, one space is inserted into AGTGATG to result in AGTGAT-G, and three spaces are inserted into GTTAG to result in -GT--TAG. A space is denoted by a minus sign (-). The two genes are now of equal length. These two strings are aligned:

AGTGAT-G
- GT--TAG

In this alignment, there are four matches, namely, G in the second position, T in the third, T in the sixth, and G in the eighth. Each pair of aligned characters is assigned a score according to the following scoring matrix.

	A	C	G	T	—
A	5	−1	−2	−1	−3
C	−1	5	−3	−2	−4
G	−2	−3	5	−2	−2
T	−1	−2	−2	5	−1
—	−3	−4	−2	−1	*

The * denotes that a space-space match is not allowed. The score of the alignment above is $(-3)+5+5+(-2)+(-3)+5+(-3)+5=9$.

Of course, many other alignments are possible. One is shown below (a different number of spaces are inserted into different positions):

AGTGATG
-GTT A -G

This alignment gives a score of $(-3)+5+5+(-2)+5+(-1)+5=14$. So, this one is better than the previous one. As a matter of fact, this one is optimal since no other alignment can have a higher score. So, it is said that the similarity of the two genes is 14.

Input

The input consists of T test cases. The number of test cases T is given in the first line of the input file. Each test case consists of two lines: each line contains an integer, the length of a gene, followed by a gene sequence. The length of each gene sequence is at least

one and does not exceed 100.

Output

The output should print the similarity of each test case, one per line.

Sample Input

```
2
7 AGTGATG
5 GTTAG
7 AGCTATT
9 AGCTTTAAA
```

Sample Output

```
14
21
```

1）**问题的描述与分析**

人类基因是由 A、C、G、T 4 种核苷构成的序列。在生物学研究中，常常需要比较两个基因序列的相似程度。本问题对两个基因序列给出一种相似程度的度量方法——联配计分。具体方法是，在序列中必要的位置上添加空格(用"－"表示)，使得两个序列长度一致，然后按给定的计分矩阵计算出序列的相似度。目标是找出相似度最大的联配。问题可形式化为如下。

输入：两个由{A，C，G，T}中的符号构成的序列 X、Y，以及计分矩阵 A。

输出：计算出序列 X、Y 的最大联配相似度。

其中，计分矩阵 A 由下表给出：

	A	C	G	T	—
A	5	−1	−2	−1	−3
C	−1	5	−3	−2	−4
G	−2	−3	5	−2	−2
T	−1	−2	−2	5	−1
—	−3	−4	−2	−1	*

其实，DNA 序列的 LCS 问题可视为上述问题中计分矩阵为单位阵时的一个特例。所以，设 $s[i,j]$为 X、Y 的前缀 X_i、Y_j 的最大联配相似度。则不难理解下列递归式：

$$s[i,j]=\begin{cases}0 & i=0 \text{ 且 } j=0\\ s[i-1,0]+A[x_i,-] & i>0 \text{ 且 } j=0\\ s[0,j-1]+A[-,y_i] & i=0 \text{ 且 } j>0\\ \max\{s[i-1,j-1]+A[x_i,y_j],s[i-1,j]+A[x_i,-],s[i,j-1]+A[-,y_i]\} & \text{其他}\end{cases}$$

利用此递归式，可以设计一个类似于 LCS-LENGTH 的算法。

2）**算法描述与分析**

```
HUMAN-GENE-FUNCTIONS(X,Y,A)
1 m←length[X]
```

```
2 n←length[Y]
3 s[0,0]←0
4 for i←1 to m
5     do s[i,0]←s[i-1,0]+A[x_i,-]
6 for j←0 to n
7     do s[0,j]←s[0,j-1]+A[-,y_j]
8 for i←1 to m
9     do for j←1 to n
10           do s[i,j]←max{s[i-1,j-1]+A[x_i,y_j],
11                         s[i-1,j]+A[x_i,-],
12                         s[i,j-1]+A[-,y_j]}
14 return s[m,n]
```

算法 8-7 计算两个 DNA 序列最大联配相似度的算法过程

算法 8-7 的运行时间主要消耗在第 8～12 行的二重嵌套 **for** 循环上。很容易看出，第 10～12 行的循环体共重复 mn 次。所以，算法过程 HUMAN-GENE-FUNCTIONS 的时间复杂度为 $\Theta(mn)$。

3）**程序实现**

由于计分矩阵是固定的，为了简化函数参数，将其定义为如下的全局二维数组。

```
int a[5][5]={5,-1,-2,-1,-3,                    /* 计分矩阵 */
             -1,5,-3,-2,-4,
             -2,-3,5,-2,-2,
             -1,-2,-2,5,-1,
             -3,-4,-2,-1,0};
```

注意：表示空格与空格比对的 a[4][4]的数据在算法中是不会用到的，所以可取任意值，此处取 0。

下面两个辅助函数 max3 及 index 是在实现算法 8-7 时分别用来计算 3 个整数项 $s[i-1,j-1]+A[x_i,y_j]$，$s[i-1,j]+A[x_i,-]$和 $s[i,j-1]+A[-,y_j]$最大值及将字符 A、C、G、T 映射到计分矩阵下标的。

```
int max3(int a,int b,int c){                   /* 计算 3 个整数的最大值 */
    int d=a>=b? a:b;
    return d>=c? d:c;
}
int index(char A){                             /* 将字符 A、C、G、T 转换为下标 0、1、2、3 */
    switch(A){
        case 'A': return 0;
        case 'C': return 1;
        case 'G': return 2;
        case 'T': return 3;
    }
}
```

程序 8-7 用来计算 3 个整数最大值及转换数组下标的 C 函数

下列函数 humanGeneFunctions 实现算法 8-7。

```
1 int humanGeneFunctions(char *x,char *y){                                  /*联配计分*/
2     int *s,i,j,m,n,r;
3     m=strlen(x);n=strlen(y);
4     assert(s=(int*)malloc((m+1)*(n+1)*sizeof(int)));
5     s[0]=0;
6     for(i=1;i<=m;i++)
7         s[i*(n+1)]=s[(i-1)*(n+1)]+a[index(x[i-1])][4];
8     for(j=1;j<=n;j++)
9         s[j]=s[j-1]+a[4][index(y[j-1])];
10    for(i=1;i<=m;i++)
11        for(j=1;j<=n;j++)
12            s[i*(n+1)+j]=max3(s[(i-1)*(n+1)+j-1]+a[index(x[i-1])][index(y[j-1])],
13                              s[(i-1)*(n+1)+j]+a[index(x[i-1])][4],
14                              s[i*(n+1)+j-1]+a[4][index(y[j-1])]);
15    r=s[m*(n+1)+n];
16    free(s);
17    return r;
18 }
```

程序 8-8　实现算法 8-7 的 C 函数

对程序 8-8 的说明如下。

(1) 由于将计分矩阵定义成全局量，所以函数 humanGeneFunctions 比伪代码过程 HUMAN-GENE-FUNCTIONS 少一个参数，仅保留表示两个 DNA 序列的串 x 和 y。

(2) 函数体内定义了与算法过程中同名的变量 m、n、s、i、j 等。虽然 s 应该是一个用来表示(m+1)×(n+1)数表的二维数组，但出于程序运行效率的考虑仍然将其定义成一个动态的一维数组(第 4 行)，并按行优先原则存储二维数组的数据。

(3) 程序的代码结构与算法过程的伪代码结构十分相似。要注意的是伪代码中计分矩阵中元素的下标是用对应的字符 A、C、G、T 及空格“－”表示的，这在 C 代码中是不行的。所以，需要调用在程序 8-7 中定义的函数 index 来将 A、C、G、T 分别转换为 0、1、2、3 而将空格(表为－)直接对应 4。

利用这些函数定义，解决 Human Gene Functions 问题的主函数设计如下。

```
int main(){
    FILE *f1=fopen("chap08/Human Gene Functions/inputdata.txt","r"),
         *f2=fopen("chap08/Human Gene Functions/outputdata.txt","w");
    char x[101],y[101];
    int t,i,a;
    assert(f1&&f2);
    fscanf(f1,"%d",&t);
    for(i=0;i<t;i++){
        fscanf(f1,"%d%s",&a,x);
        fscanf(f1,"%d%s",&a,y);
```

```
11          fprintf(f2,"%d\n",humanGeneFunctions(x,y));
12      }
13      fclose(f1),fclose(f2);
14      return 0;
15 }
```

程序 8-9 解决 Human Gene Functions 问题的 C 程序

程序中第 7 行从输入文件 f1 中读取案例数 t。第 8～12 行的 **for** 循环处理每一个案例。其中,第 9 行和第 10 行从 f1 中读取案例中的两个 DNA 串 x 和 y。第 11 行调用程序 8-8 计算这两个串的最优联配度并写入输出文件 f2 中。程序 8-7～程序 8-9 存储在文件夹 chap08/Human Gene Functions 的源文件 Human Gene Functions. c 中。

8.3 0-1 背包问题

8.3.1 问题描述

将第 7 章 7.1.1 节讨论过的 0-1 背包问题重新描述如下。设有 n 种物品,第 i 种物品的质量为 w_i,价值为 v_i,$i=1,2,\cdots,n$。给定一个背包,最多能装质量为 C 的物品。第 i 种物品或放入包中,或留在包外。问将哪些物品放到包内能使得包中所装物品总价值最大?其中,w_i、v_i 都是正整数,$i=1,2,\cdots,n$。C 也是正整数。如果用一个向量 $x=(x_1,x_2,\cdots,x_n)$ 来表示此问题的解,其中 $x_i=\begin{cases}1 & i\text{ 号物品放入包中}\\ 0 & i\text{ 号物品未放入包中}\end{cases}$,则 0-1 背包问题可形式化地描述为如下。

输入:集合 $W=\{w_1,w_2,\cdots,w_n\}$,$V=\{v_1,v_2,\cdots,v_n\}$,数值 C。

输出:向量 $x=(x_1,x_2,\cdots,x_n)$,$x_i\in\{0,1\}$,$1\leqslant i\leqslant n$,使得 $\sum x_iw_i\leqslant C$,且 $\sum x_iv_i$ 最大。

这是一个典型的组合优化问题。所有 2^n 个向量 $x=(x_1,x_2,\cdots,x_n)$,$x_i\in\{0,1\}$,$1\leqslant i\leqslant n$,构成问题的解空间。可行解受 $\sum x_iw_i\leqslant C$ 限制且以 $\sum x_iv_i$ 为目标值。目的是求出可行解中的最优解——目标值最大者。若用回溯算法,将耗费指数级的时间。

8.3.2 算法设计与分析

1. 最优子结构与子问题的重叠

下面来考察此问题是否具有最优子结构和子问题重叠性质。

定理 8-2(0-1 背包问题的最优子结构) 设 $y=(y_1,y_2,\cdots,y_i)$是 $W=\{w_1,w_2,\cdots,w_i\}$、$V=\{v_1,v_2,\cdots,v_i\}$、数值为 j 的 0-1 背包问题的一个最优解。

(1) 若 $w_i>j$,则 $y'=(y_1,y_2,\cdots,y_{i-1})$为 $W=\{w_1,w_2,\cdots,w_{i-1}\}$、$V=\{v_1,v_2,\cdots,v_{i-1}\}$、数值为 j 的子问题的最优解。

(2) 若 $w_i \leqslant j$，且 $y_i=0$，则 $y'=(y_1,y_2,\cdots,y_{i-1})$ 为 $W=\{w_1,w_2,\cdots,w_{i-1}\}$、$V=\{v_1,v_2,\cdots,v_{i-1}\}$、数值为 j 的子问题的最优解。

(3) 若 $w_i \leqslant j$，且 $y_i=1$，则 $y'=(y_1,y_2,\cdots,y_{i-1})$ 为子问题 $W=\{w_1,w_2,\cdots,w_{i-1}\}$，$V=\{v_1,v_2,\cdots,v_{i-1}\}$，数值为 $j-w_i$ 的最优解。

(1)、(2) 的结论是显然的，此处仅就(3) 展开讨论。假定 $(y_1,y_2,\cdots,y_{i-1})$ 不是子问题 $W=\{w_1,w_2,\cdots,w_{i-1}\}$，$V=\{v_1,v_2,\cdots,v_{i-1}\}$，数值为 $j-w_i$ 的最优解，且其另一个最优解为 $z=(z_1,z_2,\cdots,z_{i-1})$，即 $\sum_{k=1}^{i-1} z_k w_k \leqslant j-w_i$，且 $\sum_{k=1}^{i-1} z_k v_k > \sum_{k=1}^{i-1} y_k v_k$。于是，有 $\sum_{k=1}^{i-1} z_k w_k + y_i w_i \leqslant j$ 且 $\sum_{k=1}^{i-1} z_k v_k + y_i v_i > \sum_{k=1}^{i} y_k v_k$，即 $(z_1,z_2,\cdots,z_{i-1},y_i)$ 是原问题 $W=\{w_1,w_2,\cdots,w_i\}$，$V=\{v_1,v_2,\cdots,v_i\}$，数值为 j 的一个比 $y=(y_1,y_2,\cdots,y_i)$ 的目标值更大的解。这与 $y=(y_1,y_2,\cdots,y_i)$ 是最优解矛盾。这说明 0-1 背包问题是具有最优子结构特性的。

设 $m[i,j]$ 是子问题 $W=\{w_1,w_2,\cdots,w_i\}$，$V=\{v_1,v_2,\cdots,v_i\}$，数值为 j 的最优解的目标值，其中，$i=1,2,\cdots,n$，$j=1,2,\cdots,C$。根据最优子结构特性，有：

$$m[i,j]=\begin{cases}0 & i=0 \text{ 或 } j=0 \\ m[i-1,j] & i>0 \text{ 且 } w_i>j \\ \max\{v_i+m[i-1,j-w_i],m[i-1,j]\} & i>0 \text{ 且 } w_i \leqslant j\end{cases} \tag{8-13}$$

这是一个 $(n+1)\times(C+1)$ 矩阵 $m[0..n,0..C]$。例如，对于问题 $W=\{2,3,4,5\}$，$V=\{3,4,5,7\}$，$C=9$，可以计算出下面的矩阵 m。

i \ j	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12
4	0	0	3	4	5	7	8	10	11	12

在此例中，计算 $m[2,j]$ 时多次用到 $m[2,1]$ 的计算，如图 8-6 所示。因此，0-1 背包问题是具有子问题重叠特性的。注意，此定义中，二维数表 m 的行标和列标都是从 0 开始编号的。

2. 计算最大价值

利用 0-1 背包问题的最优子结构，并考虑其子问题的重叠性，可以写出下列自底向上计算最大价值的动态规划算法。

```
KNAPSACK(v,w,C)
1 n←length[v]
2 for j←0 to C
3       do m[0,j]←0
4 for i←1 to n
```

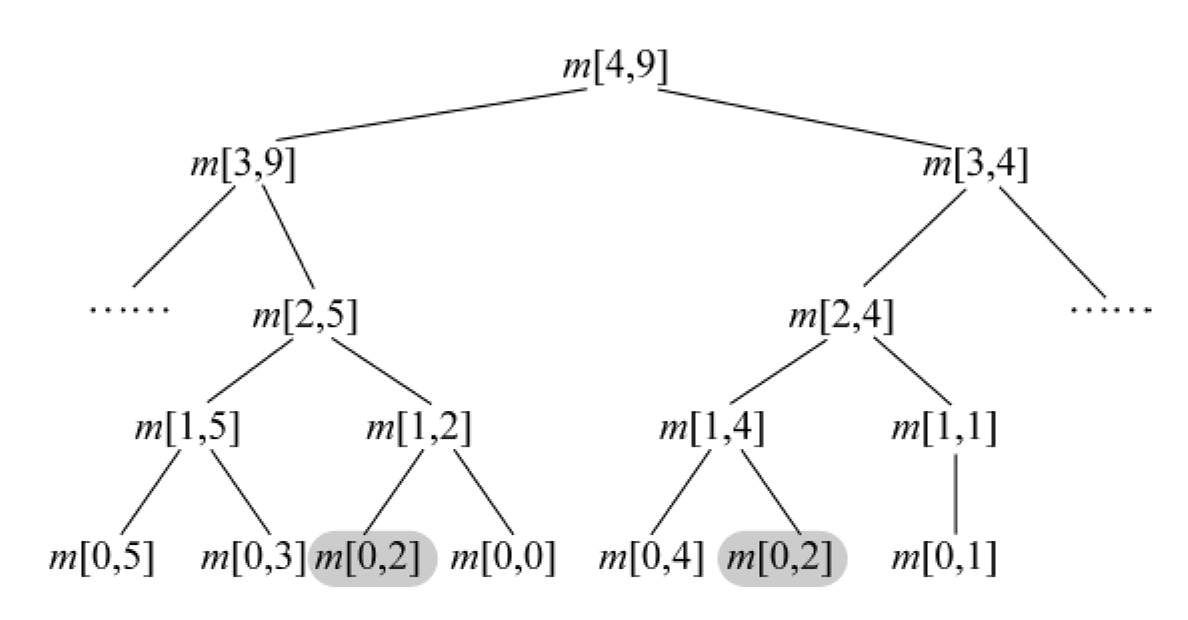


图 8-6　0-1 背包问题的子问题重叠

```
5       do m[i,0]←0
6          for j←1 to C
7              do m[i,j]←m[i-1,j]
8                    if wᵢ≤j
9                      then if vᵢ+m[i-1,j-wᵢ]>m[i-1,j]
10                              then m[i,j]←vᵢ+m[i-1,j-wᵢ]
11 return m
```

算法 8-8　计算 0-1 背包问题中最大价值的 KNAPSACK 算法

算法中的第 2 行和第 3 行的 **for** 循环，以及第 4～10 行的 **for** 循环中的第 5 行是按式(8-13)计算最底层的子问题最优解的值。

3. 构造一个最优解

利用 KNAPSACK 返回的矩阵 m，可以构造出最优解。

```
BUILD-SOLUTION(m,w,C)
1 n←length[w]
2 j←C
3 for i←n downto 1
4       do if m[i,j]=m[i-1,j]
5             then x[i]←0
6             else x[i]←1
7                   j←j-w[i]
8 return x
```

算法 8-9　利用算法 8-8 返回的矩阵构造 0-1 背包问题最优解的 BUILD-SOLUTION 算法

算法根据定理 8-2 及式(8-13)，当且仅当 $m[i,j]=m[i-1,j-w_i]+v_i$ 时，第 i 件物品放入背包内(第 6 行 x_i 置为 1)，否则必有 $m[i,j]=m[i-1,j]$，即第 i 件物品不放入背包内(第 5 行 $x[i]$置为 0)。

4. 算法的运行时间

过程 KNAPSACK 中，第 2 行和第 3 行的 **for** 循环耗时 $\Theta(n)$，第 5～10 行的两个嵌套的 **for** 循环，外层重复 n 次，里层重复 C 次，循环体内消耗常数时间，所以该过程的时间复杂度

是 $\Theta(nC)$，而过程 BUILD-SOLUTION 的时间复杂度显然是 $\Theta(n)$。

8.3.3　程序实现

由于很多应用可模型化为 0-1 背包问题，所以将算法 8-8 和算法 8-9 实现为程序是有实用价值的。

1. 计算最大价值

```
1 int *knapsack(int *w,int *v,int n,int c){
2     int *m=(int*)malloc((n+1)*(c+1)*sizeof(int)),i,j;
3     for(j=0;j<c+1;j++)
4         m[j]=0;                                      /*m[0,j]←0*/
5     for(i=1;i<=n;i++){
6         m[i*(c+1)]=0;                                /*m[i,0]←0*/
7         for(j=1;j<=c;j++){
8             m[i*(c+1)+j]=m[(i-1)*(c+1)+j];           /*m[i,j]←m[i-1,j]*/
9             if(w[i-1]<=j)
10                if(m[(i-1)*(c+1)+j-w[i-1]]+v[i-1]>m[(i-1)*(c+1)+j])
11                    m[i*(c+1)+j]=m[(i-1)*(c+1)+j-w[i-1]]+v[i-1];
12        }
13    }
14    return m;
15 }
```

程序 8-10　实现算法 8-8 KNAPSACK 过程的 C 函数

对程序 8-10 的说明如下。

(1) 与算法 KNAPSACK 相比，函数 knapsack 除了质量数组 w、价值数组 v、背包容量 c 3 个参数以外，还多一个说明物件样数的参数，也就是数组 w 及 v 的元素个数的 n。

(2) 仍然用一维数组按行优先方式存储矩阵。第 4 行为数组 m 分配存储 $(n+1)\times(c+1)$ 矩阵所需空间。

(3) 程序代码与算法伪代码的结构几乎一致，需要注意的是用一维数组表示的矩阵的元素访问方式。

2. 构造最优解

对算法 8-9 的实现如下。

```
int *buildSolution(int *m,int n,int *w,int c){
    int i,j=c;
    int *x=(int*)malloc(n*sizeof(int));
    for(i=n;i>=1;i--)
        if(m[i*(c+1)+j]==m[(i-1)*(c+1)+j])
            x[i-1]=0;
        else{
```

```
            x[i-1]=1;
            j-=w[i-1];
        }
    return x;
}
```

程序 8-11　实现算法 8-9 的 C 函数

函数 buildSolution 除了具有矩阵 m、质量数组 w 和背包容量 c 3 个参数以外,还多了一个说明物件数的 n。代码结构与算法伪代码结构一致,此处不再赘述。

程序 8-10 及程序 8-11 定义的函数存储在文件夹 dprog 中的源文件 knapsack. c 中,它们的原型声明存储于同一文件夹的头文件 knapsack. h 中,以备重用。

8.3.4　应用

1. 温馨旅程

Happy Travel

Description

May Day is coming, Ray and Amy decide to have a travel during the holiday in such a wonderful spring. Due to the long vocation, many things should be packed. Both Amy and Ray owns a bag with an infinite capacity and they come to a conclusion that all the things such as food, drink, tent will be loaded into these two bags. Caring for each other, they wanted the difference between the weights of the two bags can be minimized. Of course, Ray's bag is always not lighter than that of Amy.

Input

Input file contains multiple test cases. For each test case, an integer $N(1 \leqslant N \leqslant 100)$ appeals on the first line indicating the number of things should be loaded. The next line will be N integers no more than 100000, denoting the weight of each thing. Proceed to the end of the file.

Output

Output two integers for each test case, denoting the total weight of Amy's and Ray's bag respectively. Please output as the format in the Sample Output.

Sample Input

```
2
5 6
3
1 3 5
```

Sample Output

```
Case 1: 5 6
```

Case 2：4 5

1）**问题描述与分析**

Amy 和 Ray 外出徒步春游，携带了若干件行李，每件行李都有各自的质量（整数）。他们相互关爱，不愿让对方比自己更累，想把行李分成质量相当的两份各自携带。本题目实际上是要求一个整数集合 A（各件行李的质量）的一个划分：B_1 和 B_2（分别表示 Amy 和 Ray 携带的各件行李的质量），$B_1 \cap B_2=\varnothing$①，$B_1 \cup B_2=A$，使得 B_1 和 B_2 的元素之和间的差最小（两人相互爱护尽量不让对方比自己更累）。可以把此问题形式化为如下。

输入：整数集合 A。

输出：整数集合 B_1 和 B_2 的元素之和 w_1 与 w_2。其中，B_1 和 B_2 是 A 的一个划分，且差值 $|w_1-w_2|$ 为最小。

设 A 中物品的质量 $\{a_1,a_2,\cdots,a_n\}$ 之和为 W。若将 A 中物品的质量 $\{a_1,a_2,\cdots,a_n\}$ 同时视为这些物品的价值，并设 $W/2$ 为背包承重量 C，则可以把这个问题转化为一个 0-1 背包问题：在 $\sum\limits_{i=1}^{n} x_i a_i \leqslant C$ 的限制下，使 $\sum\limits_{i=1}^{n} x_i a_i$ 的值最大。

2）**程序实现**

利用程序 8-10 很容易写出解决 Happy Travel 问题的 C 程序。

```
int main(){
    FILE *f1=fopen("chap08/Happy Travel/inputdata.txt","r"),
         *f2=fopen("chap08/Happy Travel/outputdata.txt","w");
    int n,*w,*m,c,k=1,i;
    assert(f1&&f2);
    while(!feof(f1)){
        fscanf(f1,"%d",&n);                          /*读取行李件数*/
        assert(w=(int*)malloc(n*sizeof(int)));
        for(i=0,c=0;i<n;i++){                         /*读取每件行李的质量*/
            fscanf(f1,"%d",w+i);
            c+=w[i];                                  /*计算总质量*/
        }
        m=knapsack(w,w,n,c/2);                        /*计算不超过总质量一半的最大值*/
        fprintf(f2,"Case %d: %d %d\n",k++,m[n*(c/2+1)+c/2],c-m[n*(c/2+1)+c/2]);
        free(w);
    }
    fclose(f1),fclose(f2);
    return 0;
}
```

程序 8-12　解决 Happy Travel 问题的 C 程序

①　∅表示空集合。

对程序8-12的说明如下。

(1) 第2行和第3行分别打开输入文件f1和输出文件f2。输入文件由若干案例数据构成,每个案例包含两行数据,第1行仅含1个整数n表示有多少件行李。第2行包含n个表示每件行李质量的整数。

(2) 变量w表示保存各件行李质量的数组,m表示保存各层子问题最优解值的数表,c表示行李的总质量,k表示案例号,i为循环控制变量。

(3) 按输入文件格式,第6~16行的**while**循环对每个案例重复执行。每次重复中,第7行从f1中读取行李件数n,第9~12行的**for**循环读取n件行李的质量w[i]并计算总质量c。第13行调用程序8-10定义的函数knapsack,质量数组为w,价值数组也为w,背包承重量为c/2的背包问题计算各层子问题最优解值,返回m。m[n*(c/2+1)+c/2]表示二维数表中元素m[n,c/2],它表示最顶层问题的最优解的值,也就是质量不超过c/2的最重的行李子集的总质量,这应该归女孩Amy,男孩Ray无论如何还是应该在体力方面照顾女孩的,他带的行李质量为c−m[n*(c/2+1)+c/2]。第14行将这两个数据写入输出文件f2。

程序8-12存储在文件夹chap08/Happy Travel中的源文件HappyTravel.c中。

2. 导游的算盘

The Cicerone's Abacus

Description

Currently for the requirement of the sightseers, many cicerones often take them to shopping for some souvenirs in addition to the introduction of the local culture and historic sites. Some souvenir shops give the cicerones who bring sightseers some kickbacks in order to make them bring more. The amount of the kickback is determined by the value of the goods. The shops usually list the values of the goods and the relevant kickbacks for the cicerones to make them clear of the reward they can earn from each kind of goods.

Assume the cicerone has known the maximum amount of money every sightseer intends to spend and the trust of the sightseers on him. The sightseers will buy all souvenirs in any quantity recommended by the cicerone only if their maximum budgets are kept. Before going to the shop, the cicerone starts calculating which goods he should recommend to which sightseer so as to make himself obtain the maximum kickbacks. Unfortunately, this is such a hard problem for him that he never works it out.

Luckily, the cicerone meets you who is good at programming, so he hopes you can program to finish this job. Assume the shop has enough quantity of every goods to meet the shopping requirement of all sightseers.

Input

The input file contains multiple test cases. For each case the first line includes a positive integer n representing the number of sightseers who intend to buy souvenirs. The second line includes n integers each of which is the maximum budget (positive integer and less than 2000) each sightseer plans to spend. There is a positive integer m ($m<101$) in

the following line indicating the categories of the goods. Then comes m lines. Each line has two integers. The first is the price p of the goods ($p<1000$) and the second is the value of the relevant kickback (no more than 0.2p).

Output

For each test case output one line contains the maximum value of the kickback the cicerone can obtain.

Sample Input

```
3
200 300 300
5
12 2
20 4
30 5
50 5
100 18
```

Sample Output

```
160
```

1）问题的描述与分析

旅游纪念品商家对带领旅客来店购买商品的导游给予回扣奖励。回扣量取决于旅客们购买的商品数量。商家对 m 种纪念品的每一种都列出了价格和给导游的回扣，记 m 件物品的价格为 $P=\{p_1,p_2,\cdots,p_m\}$，回扣 $K=\{k_1,k_2,\cdots,k_m\}$。导游知道每个游客最多能花多少钱购买纪念品，n 个游客的购物预算为 $C=\{c_1,c_2,\cdots,c_n\}$。每种物品货源丰富，游客可在预算条件下，按导游的建议任意购买。问题是导游应建议各位游客购买哪些物品，买多少能使自己得到的回扣最多？

对任何游客 j 而言，类似一个 0-1 背包问题：每种纪念品的价格 $P=\{p_1,p_2,\cdots,p_m\}$构成质量数组，每种纪念品对应的回扣额构成价值数组 $K=\{k_1,k_2,\cdots,k_m\}$，而游客的预算额 c_j 视为背包承重量。然而，这个模型与 0-1 背包模型的不同之处在于：0-1 背包问题中，每件物品要么放入背包，要么留下，而在本问题中，一种纪念品可以被游客购买 1 件、2 件、……，仅受游客预算额 c_j 的限制。这样，需要设法将本问题的模型转化为 0-1 背包模型。

我们可以想象，旅游商店为方便顾客，将第 i 种纪念品分成 1 件包装，2 件包装，……，最多 x 件包装，x 是满足 $x(x+1)p_i/2\leqslant c_j$ 的最大值[①]。这样，这些包装好的东西可以视为 0-1 背包模型中的物品了，有确定的价值，确定的回扣额，要么留下，要么装入包中。

确切地说，对第 j 号游客，将第 i 件物品扩展为 m_i"件" 物品，其价格为$\{p_i,2p_i,3p_i,\cdots,m_ip_i\}$，回扣为$\{k_i,2k_i,3k_i,\cdots,m_ik_i\}$。其中 $m_i=\max\{x \mid x\in N, x(x+1)p_i/2\leqslant c_j\}$，$i=1,2,\cdots,m$。也就是说把 1 件 i 号物品，2 件 i 号物品，……，m_i 件 i 号物品视为新的 m_i 件

① 对第 i 种纪念品，游客 j 购买 1 件包装的需付 p_i 元，购买 2 件包装的需付 $2p_i$ 元，……，他(/她)可以购买这种纪念品受 $p_i+2p_i+3p_i\cdots+xp_i\leqslant c_j$ 的限制，而 $p_i+2p_i+3p_i\cdots+xp_i=x(x+1)p_i/2$，故最多可购买的量受 $x(x+1)=p_i/2\leqslant c_j$ 的限制。

物品,而 m_i 是游客 j 最多能购买的 i 号物品数。把 $\bigcup_{i=1}^{m}\{p_i, 2p_i, 3p_i, \cdots, m_ip_i\}$ 中的 $m'=\sum_{i=1}^{m} m_i$ 个新的物品的价格记为 $\{p'_1, p_2', \cdots, p_{m'}'\}$,把 $\bigcup_{i=1}^{m}\{k_i, 2k_i, 3k_i, \cdots, m_ik_i\}$ 中的 m' 个新的物品的回扣记为 $\{k'_1, k'_2, \cdots, k'_{m'}\}$,则可将一个游客的购物问题可转化为0-1背包问题:在 $\sum_{i=1}^{m'} x_ip'_i \leqslant c_j$ 的限制下,最大化 $\sum_{i=1}^{m'} x_ik'_i$。

每个游客对应的子问题都一一解决,所得的最优值之和就是原问题的最优值。

2) **算法的伪代码描述**

将上述算法思想写成如下的伪代码过程。

```
THE-CICERONE-ABACUS(p,k,C)
1 max←0
2 n←length[C]
3 for j←1 to n
4     do (p1,k1)←EXTEND(p,k,C[j])
5        m←length[p1]
6        matrix←KNAPSACK (p1,k1,C[j])
7        max←max+matrix[m,C[j]]
8 return max
```

算法 8-10 解决 The Cicerone's Abacus 问题的算法 THE-CICERONE-ABACUS 过程

其中,第 4 行调用下列 EXTEND 过程对第 j 个游客创建 0-1 背包模型。

```
EXTEND(p,k,C)
m←length[p]
newp←newk←∅
for i←1 to m
    do m1←max{x|x∈N,x(x+1)pi/2≤C}
     for j←1 to m1
         do append j×pi to newp
            append j×ki to newk
return newp and newk
```

算法 8-11 将 1 个游客的购物数据创建为 0-1 背包模型的 EXTEND 过程

3) **程序实现**

算法 8-11 将返回两个数组对象 *newp* 和 *newk*,这对于 C 函数来说是办不到的。一个可行的解决办法是将两个对象整合到一个结构体中。这种情形在应用中会经常遇到,函数需要返回的多个对象也不限于整型数组,可以是各种类型的数据。因此,定义如下通用的序偶数据类型 pair。

```
typedef struct {                          /* 用来存储两个指针的结构体 */
    void * first;
    void * second;
```

```
}pair;
```

并定义一个根据两个指针数据创建 pair 对象的函数 make_pair。

```
pair make_pair(void *f,void *d){
    pair p={f,d};
    return p;
}
```

将此类型定义及函数 make_pair 的原型声明存储在文件夹 utility 中的头文件 pair.h 中，而将函数 make_pair 的定义存储在同一文件夹内的源文件 pair.c 中，以备重用。

做了这些准备工作后，先来实现算法 8-11。

```
pair extend(int *p,int *k,int *m,int C){
    int *newp,*newk,*m1,newm,x,i,j,t;
    assert(m1=(int*)malloc((*m)*sizeof(int)));
    for(i=0,x=1;i<*m;i++){
        while(x*(x+1)*p[i]/2<C)x++;       /*计算 max{x|x∈N,x(x+1)p_i/2≤c_j}*/
        m1[i]=x;
    }
    for(i=0,newm=0;i<*m;i++)              /*计算 newp 和 newk 的长度*/
        newm+=m1[i];
    assert(newp=(int*)malloc(newm*sizeof(int)));
    assert(newk=(int*)malloc(newm*sizeof(int)));
    for(i=0,t=0;i<*m;i++)                 /*计算数组 newp 和 newk*/
        for(j=0;j<m1[i];j++,t++){
            newp[t]=(j+1)*p[i];
            newk[t]=(j+1)*k[i];
        }
    free(m1);
    *m=newm;                              /*m 带回 newp 和 newk 的长度*/
    return make_pair(newp,newk);
}
```

程序 8-13　实现算法 8-11 的 C 函数

对程序 8-13 的说明如下。

（1）与算法过程相比，函数 extend 除了表示纪念品价格的数组 p、回扣额数组 k 和游客预算额 C 以外，还多了一个表示数组 p 和 k 的元素个数的参数 m。该函数的任务就是用 p、k、C 创建一个合适的 0-1 背包模型，包括质量数组和价值数组。所以，返回值类型是上述的 pair。此外，由于 C 语言的数组不含有数组长度的属性，所以将表示 p、k 的长度的参数 m 定义成指针类型，这样既可以作为表示 p、k 的长度，也可向外部传递新的数组 newp 和 newk 的长度。

（2）函数体内声明了作为返回对象的数组 newp 和 newk，用来暂存各种纪念品最多包装数的数组 m1。newp 和 newk 的长度需在运行时确定，所以用指针来表示。虽然 m1 的长度为传递进来的参数 m 的值，但 C 语言数组定义中元素个数需为常量，故不得已也将其定

义为指针。

(3) 函数代码的结构与算法过程的结构也略有不同。算法中 newp 和 newk 可视为可变长线性表,而是将两者定义成由指针指引的数组。所以,必须先计算出它们的长度。第 4～7 行的 **for** 循环对每一种商品 i 计算最大的 x,使得 $x(x+1)\ p_i/2\leqslant C$(第 5 行的 **while** 循环完成),并将值保存在 m1[i]中。第 8 行和第 9 行的 **for** 循环累加数组 m1 中的元素,得到数组 newp 和 newk 的长度 newm。第 12～16 行的两重 **for** 嵌套循环对应算法过程中的第 3～7 行的 **for** 嵌套循环,计算 newp 和 newk 个元素的值。第 19 行将此两者整合与一个 pair 对象中座位函数值返回。注意,此前的第 18 行将 newm 赋值给 m 指向动态变量,以此方式将 newp 和 newk 的长度带出。

利用 extend 函数和程序 8-10 定义的函数 knapsack,将算法 8-10 实现为如下函数。

```
1  int theCiceronesAbacus(int *p,int *k,int m,int *C,int n){
2      int max=0,j,*p1,*k1,m1,*matrix;
3      pair r;
4      for(j=0; j<n; j++){
5          m1=m;
6          r=extend(p,k,&m1,C[j]);             /*将 p、k、C[j]创建为 0-1 背包模型*/
7          p1=r.first; k1=r.second;
8          matrix=knapsack(p1,k1,m1,C[j]);
9          max+=matrix[m1*(C[j]+1)+C[j]];
10         free(p1),free(k1),free(matrix);
11     }
12     return max;
13 }
```

程序 8-14　实现算法 8-9 的 C 函数

对程序 8-14 的说明如下。

(1) 与算法过程相比,函数 theCiceronesAbacus 多了两个参数:数组 p 和 k 的长度 m、数组 C 的长度 n。该函数以一个案例的最优解的值——导游对一批游客购买纪念品所得的最多回扣(整数值)。

(2) 由于 extend 函数的第 3 个参数 m 一身兼二任:向函数传递数组 p 和 k 的长度,向外部传递扩展后的数组 newp 和 newk 的长度,所以需要新设置一个变量 m1 来扮演这一角色。其他变量的设置与算法过程中的同名同意。

(3) 函数的代码结构与算法过程的结构十分接近,需要注意的是用 pair 类型整合了 extend 返回的两个数组对象,所以用第 7 行将它们还原成 p1、k1。此外,函数 knapsack(见程序 8-10)返回的 matrix 是按行优先原则存储在一维数组中的矩阵,所以第 9 行访问 matrix[m1*(C[j]+1)+C[j]]相当于访问 matrix[m1,C[j]]。

利用程序 8-13 和程序 8-14 解决 The Cicerone's Abacus 问题的 main 函数设计如下。

```
int main(){
    FILE *f1=fopen("chap04/The cicerone's abacus/inputdata.txt","r"),
         *f2=fopen("chap04/The cicerone's abacus/outputdata.txt","w");
    int *p,*k,*C,m,n,i;
```

```
5    assert(f1&&f2);
6    while(!feof(f1)){
7        fscanf(f1,"%d",&n);
8        assert(C=(int *)malloc(n * sizeof(int)));
9        for(i=0;i<n;i++)
10           fscanf(f1,"%d",C+i);
11       fscanf(f1,"%d",&m);
12       assert(p=(int *)malloc(m * sizeof(int)));
13       assert(k=(int *)malloc(m * sizeof(int)));
14       for(i=0;i<m;i++)
15           fscanf(f1,"%d %d",p+i,k+i);
16       fprintf(f2,"%d\n",theCiceronesAbacus(p,k,m,C,n));
17       free(p),free(k),free(C);
18   }
19   fclose(f1),fclose(f2);
20   return 0;
21 }
```

程序 8-15　解决 The Cicerone's Abacus 问题的 C 程序

程序中第 6～18 行的 **while** 循环处理输入文件中的每一个案例。其中，第 7 行读取本案例中的游客数 n。第 8～10 行读取读取每个游客的预算金额 C[i]。第 11 行为本案例中的商品种数 m。第 12～15 行读取每一种商品的价格 p[i]及回扣 k[i]。第 16 行调用程序 8-14 定义的函数 theCiceronesAbacus，计算导游得到的最大回扣值，并写入输出文件 f2 中。

程序 8-13～程序 8-15 存储在文件夹 chap08/The Cicerone's Abacus 中的源文件 The cicerone'sabacus. c 中。

8.4　带权有向图中任意两点间的最短路径

8.4.1　问题描述

设 $G=\langle V,E\rangle$为一有向图，n 个顶点用前 n 个正整数编号，即 $V=\{1,2,\cdots,n\}$。图 G 的每一条边$(i,j)\in E$，对应一个非负距离值 $w[i,j]$。若$(i,j)\notin E$，则 $w[i,j]=\infty$。我们约定，对每一个 $i\in V$，$w[i,i]=0$。这样，一个带权有向图 $G=\langle V,E\rangle$可以用一个 $n\times n$ 的矩阵 W 加以表示，如图 8-7 所示。

我们的问题是要找出图 G 中的每一个顶点到其他所有顶点的距离。此处，顶点 i、j 间的距离定义为从 i 出发到 j 的最短路径长度。这是一个组合优化问题，从 i 出发到 j 可能有若干条路径，每条路径都有其长度，目标是找到 i 到 j 长度最短的路径。问题形式化为如下。

输入：表示有向带权图 $G=\langle V,E\rangle$的 $n\times n$ 矩阵 $\mathbf{W}$。

输出：对任意的 i、$j\in V$，i 到 j 间的距离及最短路径。

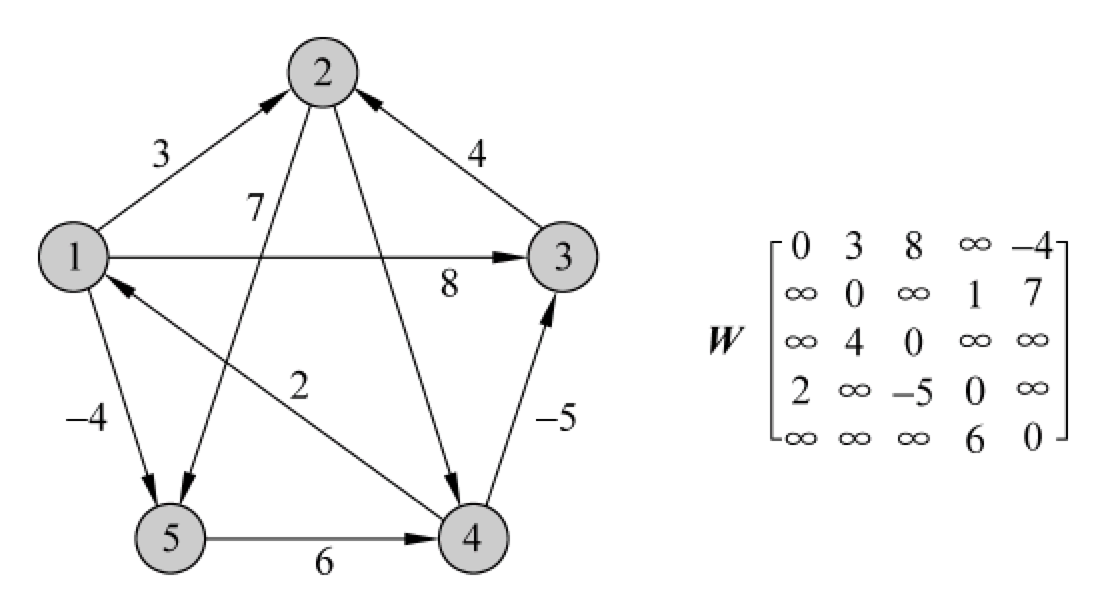


图 8-7　一个有向带权图

8.4.2　算法设计与分析

1. 最优子结构

用动态规划策略来解决这个问题。首先需要考虑问题的最优子结构特性,把它总结成如下定理。

定理 8-3　设 $G=<V,E>$的两个不同的顶点 i、$j\in V=\{1,2,\cdots,n\}$,p 是从 i 到 j 其间仅经过{ 1,2,…,k}的最短路径。

(1) 若 p 不经过顶点 k,则 p 也是从 i 到达 j 其间仅经过{1,2,…,$k-1$}的最短路径。

(2) 若 p 经过顶点 k,即 p: i　k　j。我们把前半段 i　k 记为 p_1,后半段 k　j 记为 p_2(见图 8-8)。则 p_1 是从 i 到 k 其间仅经过{1,2,…,$k-1$}的最短路径,p_2 是 k 到 j 其间不经过{1,2,…,$k-1$}的最短路径。

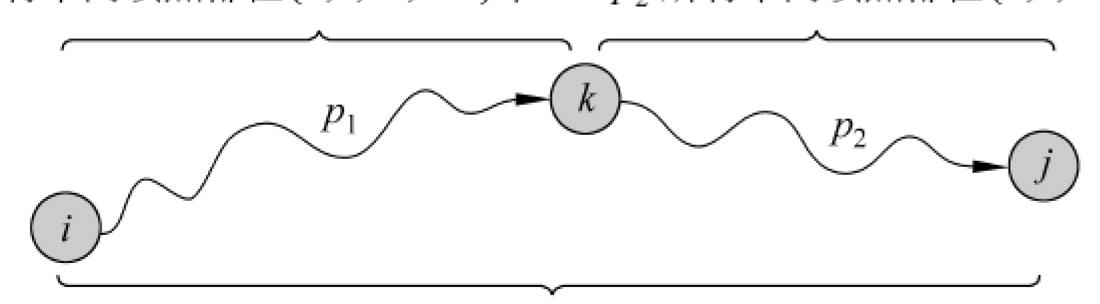


图 8-8　最短路径的最优子结构

证明　(1)的结论是显然的,此处仅就(2)展开讨论。假定 p_1 不是从 i 到 k 其间仅经过{1,2,…,$k-1$}的最短路径,而从 i 到 k 其间仅经过{1,2,…,$k-1$}的最短路径为 p_1',则在 p 中将 p_1 置换成 p_1',将得到一条比 p 更短的从 i 到达 j 其间仅经过{1,2,…,k}的路径 p'。这与 p 是从 i 到 j 其间仅经过{1,2,…,k}的最短路径矛盾。同理,可得对 p_2 的结论。

定义 $d_{i,j}^{k}$为 i 到 j 且其间仅通过{1,2,…,k}的最短路径长度。显然,$d_{i,j}^{0}$为 i 到 j 的权,而 $d_{i,j}^{n}$为 i 到 j 的最短路径长度。根据上述的任意点对间最短路径问题的最优子结构特性,我们有关于 $d_{i,j}^{k}$的递归式:

$$d_{i,j}^{k}=\begin{cases} w[i,j] & k=0 \\ \min\{d_{i,j}^{k-1},d_{i,k}^{k-1}+d_{k,j}^{k-1}\} & 1\leqslant k\leqslant n \end{cases} \qquad (8\text{-}14)$$

由此递归式的定义可知子问题具有重叠性。记:

$$D_k = (d_{ij}^k)_{n\times n},\quad k = 0,1,\cdots,n$$

则 D_n 将记录 G 的所有顶点对 (i,j) 的距离。

2. 计算任意点对的距离

利用上述关于 $d_{i,j}^k$ 的递归式，用自底向上、记表备查的方式计算各个 D_k 的算法如下。

```
FLOYD-WARSHALL(W)
1 D0←W
2 for k←1 to n
3     do for i←1 to n
4         do for j←1 to n
5             if Dk-1[i,j]<Dk-1[i,k]+Dk-1[k,j]
6                 then Dk[i,j]←Dk-1[i,j]
7                 else Dk[i,j]←Dk-1[i,k]+Dk-1[k,j]
8 return Dn
```

算法 8-12　计算带权有向图所有顶点对距离的 FLOYD-WARSHALL 算法

算法中第 1 行按式(8-14)中 $k=0$ 的条件计算 D_0。第 2～7 行按式(8-14)中 $1<k\leqslant n$ 的条件，自底向上计算 D_k 中的元素，此算法称为 Floyd 算法。显然，算法的时间复杂度为 $\Theta(n^3)$。

由于 FLOYD-WARSHALL 是以自底向上的方式计算矩阵 D_k，且计算时，仅用到 D_{k-1} 的数据，所以可以仅保留当前矩阵用来计算下一步矩阵而省略更早的计算结果，将算法改进成如下形式。

```
FLOYD-WARSHALL(W)
D←W
for k←1 to n
    do for i←1 to n
        do for j←1 to n
            if D[i,j]>D[i,k]+D[k,j]
                then D[i,j]←D[i,k]+D[k,j]
return D
```

算法 8-13　算法 8-12 的改进版本

3. 构造一个最优解

为了构造最优解——任意顶点对 (i,j) 的最短路径，必须记录路径上的顶点序列，用跟踪路径上每一个顶点的前驱来达到这一目的：$\pi_{ij}^{(k)}$ 定义为从顶点 i 到 j 的中间顶点均在集合 $\{1,2,\cdots,k\}$ 中的最短路径上顶点 j 的前驱。则

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & i = j \text{ 或 } w_{ij} = \infty \\ i & i \neq j \text{ 且 } w_{ij} < \infty \end{cases} \tag{8-15}$$

对 $k\geqslant 1$，

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & d_{ij}^{(k-1)}\leqslant d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\\ \pi_{kj}^{(k-1)} & d_{ij}^{(k-1)}> d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases} \tag{8-16}$$

那么 $\pi_{ij}^{(n)}$ 就是顶点对(i,j)的最短路径中 j 的前驱顶点。可以进一步修改算法 8-13,使得它能够同时计算出矩阵 $\Pi=(\pi_{ij}^{(n)})_{n\times n}$(见图 8-9)。

$$\boldsymbol{D}^{(0)}=\begin{bmatrix}0 & 3 & 8 & \infty & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & \infty & \infty\\ 2 & \infty & -5 & 0 & \infty\\ \infty & \infty & \infty & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(0)}=\begin{bmatrix}\text{NIL} & 1 & 1 & \text{NIL} & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2\\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL}\\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL}\\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}\end{bmatrix}$$

$$\boldsymbol{D}^{(1)}=\begin{bmatrix}0 & 3 & 8 & \infty & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & \infty & \infty\\ 2 & 5 & -5 & 0 & -2\\ \infty & \infty & \infty & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(1)}=\begin{bmatrix}\text{NIL} & 1 & 1 & \text{NIL} & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2\\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL}\\ 4 & 1 & 4 & \text{NIL} & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}\end{bmatrix}$$

$$\boldsymbol{D}^{(2)}=\begin{bmatrix}0 & 3 & 8 & 4 & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & 5 & 11\\ 2 & 5 & -5 & 0 & -2\\ \infty & \infty & \infty & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(2)}=\begin{bmatrix}\text{NIL} & 1 & 1 & 2 & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2\\ \text{NIL} & 3 & \text{NIL} & 2 & 2\\ 4 & 1 & 4 & \text{NIL} & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}\end{bmatrix}$$

$$\boldsymbol{D}^{(3)}=\begin{bmatrix}0 & 3 & 8 & 4 & -4\\ \infty & 0 & \infty & 1 & 7\\ \infty & 4 & 0 & 5 & 11\\ 2 & -1 & -5 & 0 & -2\\ \infty & \infty & \infty & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(3)}=\begin{bmatrix}\text{NIL} & 1 & 1 & 2 & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2\\ \text{NIL} & 3 & \text{NIL} & 2 & 2\\ 4 & 3 & 4 & \text{NIL} & 1\\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}\end{bmatrix}$$

$$\boldsymbol{D}^{(4)}=\begin{bmatrix}0 & 3 & -1 & 4 & -4\\ 3 & 0 & -4 & 1 & -1\\ 7 & 4 & 0 & 5 & 3\\ 2 & -1 & -5 & 0 & -2\\ 8 & 5 & 1 & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(4)}=\begin{bmatrix}\text{NIL} & 1 & 4 & 2 & 1\\ 4 & \text{NIL} & 4 & 2 & 1\\ 4 & 3 & \text{NIL} & 2 & 1\\ 4 & 3 & 4 & \text{NIL} & 1\\ 4 & 3 & 4 & 5 & \text{NIL}\end{bmatrix}$$

$$\boldsymbol{D}^{(5)}=\begin{bmatrix}0 & 1 & -3 & 2 & -4\\ 3 & 0 & -4 & 1 & -1\\ 7 & 4 & 0 & 5 & 3\\ 2 & -1 & -5 & 0 & -2\\ 8 & 5 & 1 & 6 & 0\end{bmatrix}\quad \boldsymbol{\Pi}^{(5)}=\begin{bmatrix}\text{NIL} & 3 & 4 & 5 & 1\\ 4 & \text{NIL} & 4 & 2 & 1\\ 4 & 3 & \text{NIL} & 2 & 1\\ 4 & 3 & 4 & \text{NIL} & 1\\ 4 & 3 & 4 & 5 & \text{NIL}\end{bmatrix}$$

图 8-9 对图 8-7 中的带权有向图运行算法 8-14 过程中的计算结果

```
FLOYD-WARSHALL(W)
for i←1 to n
    do for j←1 to n
        do if i=j or W[i,j]=∞
            then Π[i,j]←NIL
            else Π[i,j]←i
D←W
for k←1 to n
    do for i←1 to n
        do for j←1 to n
```

```
10            if D[i,j]>D[i,k]+D[k,j]
11               then D[i,j]←D[i,k]+D[k,j]
12                    Π[i,j]←Π[k,j]
7 return D,Π
```

算法 8-14 能在线计算前驱矩阵的 FLOYD-WARSHALL 算法

算法的第 1～5 行按式(8-15)将 Π 初始化为 Π_0，第 6 行将 D 初始化为 D_0，第 7～12 行按式(8-14)及式(8-16)自底向上地计算 D_k 及 Π_k。由于第 1～5 行的 **for** 循环仅耗时 $\Theta(n^2)$，故算法的运行时间仍为 $\Theta(n^3)$。

利用算法 8-14 返回的前驱矩阵 Π，可以构造出顶点对(i,j)的最短路径。

```
PRINT-ALL-PAIRS-SHORTEST-PATHS(Π,i,j)
if i=j
   then print i
   else if Π[i,j]=NIL
          then print "no path from" i "to" j "exists"
          else PRINT-ALL-PAIRS-SHORTEST-PATHS(Π,i,Π[i,j])
               print j
```

算法 8-15 打印顶点对(i,j)的最短路径的算法

由于一条简单路径最多含有 n 个顶点，所以最多进行 n 次递归。因此该算法的时间复杂度为 $\Theta(n)$。

8.4.3 程序实现

有向带权图顶点间的最短距离是很多应用的模型，实现 Floyd 算法是有实际意义的。

1. 计算任意点对距离

在 C 语言中可以如下实现算法 8-14。

```
pair floydWarshall(double * w,int n){
    double *d=(double *)malloc(n * n * sizeof(double));
    int i,j,k, * pi=(int * )malloc(n * n * sizeof(int));
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(j==i || w[i * n+j]>=DBL_MAX)          /* if i=j or W[i,j]=∞ */
                pi[i * n+j]=-1;                       /* p[i,j]←NIL */
            else
                pi[i * n+j]=i;                        /* p[i,j]←i */
    memcpy(d,w,n * n * sizeof(double));               /* D←W */
    for(k=1;k<=n;k++)
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                if(d[i * n+j]>d[i * n+k-1]+d[(k-1) * n+j]){
```

```
15                  d[i * n+j]=d[i * n+k-1]+d[(k-1) * n+j];
16                  pi[i * n+j]=pi[(k-1) * n+j];
17              }
18    return make_pair(d,pi);
19 }
```

程序 8-16 实现算法 8-14 的 C 函数

对程序 8-16 的说明如下。

(1) 函数 floydWarshall 除了表示图 *G* 的矩阵 w 作为参数以外,还多了一个表示图 *G* 的顶点个数的参数 n。和前两节一样,用一维数组来存储这个表示图的矩阵。由于算法要返回两个矩阵 *D* 和 Π,所以需要在 8.3.3 节中开发的数据类型 pair 作为返回值类型,其中封装了计算所得的矩阵 d 和 pi。

(2) 第 2 行和第 3 行声明了两个一维数组 d 和 pi 并为它们分配了存储空间。它们分别表示 FLOYD-WARSHALL 算法中的矩阵 *D* 和 Π。

(3) floydWarshall 函数体中的第 4～17 行对应于 FLOYD-WARSHALL 算法中第 1～12 行的操作。需要注意的是 C 代码和伪代码中关于数组下标起点值的区别,以及用一维数组表示的矩阵元素的访问形式。注意第 6 行 **if** 语句的检测条件中借用 DBL_MAX 表示∞,该常量表示 double 型数据的最大值,定义于头文件 float.h。由于检测的是双精度浮点数据的相等关系,所以用的是>=运算符。第 18 行调用函数 make_pair 将计算而得的矩阵 d 和 pi 封装到 pair 对象中,并将其返回。

2. 构造最优解

利用 floydWarshall 计算并返回的矩阵 pi,可实现构造最优解的算法 8-15。

```
1 void printAllPairsShortestPath(int * pi,int n,int i,int j){
2    if(i==j){
3        printf("%d ",i+1);
4        return;
5    }
6    if(pi[i * n+j]==-1)                              /* if p[i,j]=NIL */
7        printf("no path from %d to %d exists.\n",i+1,j+1);
8    else{
9        printAllPairsShortestPath(pi,n,i,pi[i * n+j]);
10       printf("%d ",j+1);
11   }
12 }
```

程序 8-17 实现算法 8-15 的 C 函数

对程序 8-17 的说明如下。

(1) 与算法过程相比,该函数除了以矩阵 pi、当前路径端点 i 和 j 作为参数以外,还多了一个表示 pi 的阶数的参数 n。函数直接显示最优解,而无返回值。

(2) 函数体内的代码结构与算法过程的伪代码十分接近,需要注意的是由于 pi 是整型数组,其中的元素表示图中顶点编号 1～n,故用-1 表示算法过程中的 NIL(第 6 行)。此

外，图中顶点编号与数组下标对应，而 C 语言的数组下标从 0 开始，所以第 3 行、第 6 行输出的顶点编号为 i+1 和 j+1。

程序 8-16 和程序 8-17 定义的函数存储在文件夹 dprog 中的源文件 floyd. c 中，它们的原型声明存储于同一文件夹中的头文件 floyd. h 中，以备重用。

8.4.4　应用——牛牛聚会

Bronze Cow Party

Description

One cow from each of N farms ($1 \leqslant N \leqslant 1000$) conveniently numbered 1.. N is attending the big cow party to be held at farm ＃X ($1 \leqslant X \leqslant N$). Each of M ($1 \leqslant M \leqslant 100,000$) bidirectional roads connects one farm to another. It is always possible to travel from one farm to another on the roads. Traversing road i requires T_i ($1 \leqslant T_i \leqslant 100$) units of time. One or more pairs of farms might be directly connected by more than one road.

After all the cows gathered at farm ＃X, they realized that every single cow left her party favors back at the farm. They decided to suspend the party and send all the cows back to get the party favors. The cows all travel optimal routes to their home farm and back to the party. What is the minimum number of time units the party must be suspended?

Input

* Line 1: Three space-separated integers, respectively: N, M, and X.
* Lines 2..$M+1$: Line $i+1$ describes road i with three space-separated integers, respectively: A_i, B_i, and T_i. The described road connects A_i and B_i and requires T_i time units to traverse.

Output

* Line 1: One integer: the minimum amount of time the party must be suspended.

Sample Input

```
4 8 2
1 2 7
1 3 8
1 4 4
2 1 3
2 3 1
3 1 2
3 4 6
4 2 2
```

Sample Output

```
6
```

1. 问题描述与分析

n 个来自不同的农场的牛牛相聚在牛牛 x 的家中。它们发现没带 Party 礼物，只好各

自回家去取。问晚会至少要延迟多久。n 个牛牛所在的农场可视为图中的 n 个顶点。m 条农场间道路及沿路行走所需时间可视为顶点间的边及其所具有的权值。值得注意的是,所有的道路都是双向的。所以,若输入数据中两个农场间有两条并行的道路,则应选择费时少的那一条,如图 8-10 所示。牛牛 i 从顶点 x 出发回到顶点 i 有一条最短路径,且从顶点 i 返回顶点 x 也应当有一条最短路径,两者所用时间之和是牛牛 i 回到派对所需最短时间。问题是要求出所有的牛牛按最短时间往返所用时间的最大者。

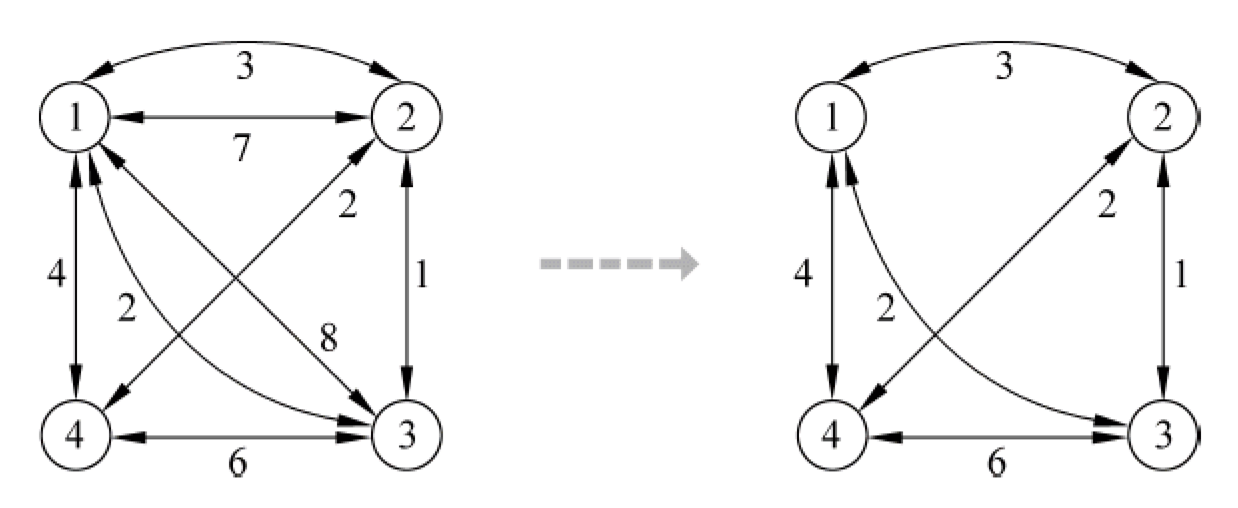


图 8-10　输入样本对应的带权有向图

问题可表示为如下。

输入:表示带权 w 的有向图 $G=<V,E>$的矩阵 W,顶点 x。

输出:$\max_{1\leqslant i\leqslant n}\{y_i \mid y_i=$从顶点 x 到顶点 i 所需最短时间+从顶点 i 到顶点 x 所需最短时间$\}$。

2. 算法描述

显然,可以利用算法 8-14 的 FLOYD-WARSHALL 算法(或算法 8-11,因为无须计算矩阵 Π)计算出任意顶点对的距离矩阵 D。D 的第 x 行表示各个牛牛从 x 回到各自农场所需最短时间,D 的第 x 列表示各个牛牛从自己的农场把礼物带回 Party 所需的最短时间,将两者对应元素相加得到各个牛牛往返所需时间,其中的最大者就是问题所求。

```
BRONZE-COW-PARTY(W,x)
D←FLOYD-WARSHALL(W)
for i←1 to n
    do time[i]←D[x,i]+D[i,x]
return SELECT(time,1,n,n)
```

算法 8-16　解决 Bronze Cow Party 问题的算法过程

其中,过程 SELECT(A,p,r,i)是第 3 章 3.2.3 节引入的选择序列 $A[p..r]$中第 i 大元素算法 3-8。

3. 程序实现

利用程序 8-12 的函数 floydWarshall,可将算法 8-16 实现如下。

```
double bronzeCowParty(double * w,int n,int x){
    double * time, * d,result;
    int i;
    pair p;
```

```
    assert(time=(double *)malloc(n * sizeof(double)));
    p=floydWarshall(w,n);
    d=p.first;
    for(i=0;i<n;i++)
        time[i]=d[i * n+(x-1)]+d[(x-1) * n+i];
    free(d);
    result= *(double *)(select(time,sizeof(double),0,n-1,n-1,doubleGreater));
    free(time);
    return result;
}
```

程序 8-18　实现算法 8-16 的 C 函数

对程序 8-18 的说明如下。

(1) 与算法过程相比,函数 bronzeCowParty 除了表示图的矩阵的数组 w 和举办聚会的牛牛编号 x 作为参数外,还多了一个表示图中顶点数的参数 n。本来问题中涉及的数据都是整型的,但是程序中需调用 floydWarshall 函数,按程序 8-16 中的定义,该函数返回的数组对象 d 是浮点型的。所以,用 d 中元素计算所得的本函数的返回值就是浮点型的。

(2) 函数中定义了与算法过程中同名同意的数组 time 和 d,但由于函数 floydWarshall 返回的是 pair 型对象,所以还需声明一个接收 floydWarshall 返回值的 pair 型变量 p。

(3) 函数题中的代码结构与算法过程的代码结构几乎一致。需要注意的是,与算法过程直接返回 SELECT 返回值不同,函数中用变量 result 接收 select 的返回值,这是因为,需要在函数运行结束前释放动态数组 time 的空间。

利用程序 8-18 写出下列解决 Bronze Cow Party 问题的程序。

```
int main(){
    FILE *f1=fopen("chap08/Bronze Cow Party/inputdata.txt","r"),
        *f2=fopen("chap08/Bronze Cow Party/outputdata.txt","w");
    int n,m,x,t,a,b,i,j;
    double *w;
    assert(f1&&f2);
    fscanf(f1,"%d %d %d",&n,&m,&x);          /* 读取顶点数 n、边数 m 和顶点编号 x */
    assert(w=(double *)malloc(n * n * sizeof(double)));
    for(i=0;i<n;i++)                          /* 初始化权矩阵 */
        for(j=0;j<n;j++)
            if(i==j)
                w[i * n+j]=0.0;
            else
                w[i * n+j]=DBL_MAX;
    for(i=0;i<m;i++){
        fscanf(f1,"%d %d %d",&a,&b,&t);       /* 读取顶点 a、b 及 a、b 间的权 t */
        if(w[(a-1) * n+b-1]>t)
            w[(a-1) * n+b-1]=w[(b-1) * n+(a-1)]=t;
    }
    fprintf(f2,"%.0f\n", bronzeCowParty(w,n,x));
```

```
21    free(w);
22    fclose(f1),fclose(f2);
23    return 0;
24 }
```

程序 8-19　解决 Bronze Cow Party 问题的程序

对程序 8-19 的说明如下。

(1) 第 7 行从输入文件中读取图中顶点个数 n、图中边数 m 和指定顶点编号 x。第 8 行用读到的 n 为表示图的权矩阵的数组 w 分配空间。

(2) 第 9～14 行的 **for** 循环将 w 初始化为如下形式:

$$\begin{pmatrix} 0 & \infty & \cdots & \infty \\ \infty & 0 & \cdots & \infty \\ \cdots & \cdots & \ddots & \cdots \\ \infty & \infty & \cdots & 0 \end{pmatrix}$$

第 15～19 行的 **for** 循环从输入文件中读取 m 条边的信息,并用其创建图的权矩阵 w。其中,第 17 行和第 18 行的 **if** 语句对满足读到的边(a,b)之权值 t 小于 w[a,b],则将 w[a,b]的值改写为 t。由于之前将矩阵非对角线上的元素初始化为∞,所以第一次处理的边信息一定会更新 w[a,b],若 a、b 间有两条平行边,则按此 **if** 语句的条件,必将维持权较小的那条边。

(3) 第 20 行将调用 bronzeCowParty 函数得到的值写到输出文件中。

程序 8-18 和程序 8-19 定义的函数存储在文件夹 chap08/Bronze Cow Party 中的源文件 BronzeCowParty.c 中,读者可打开研读并运行验证。

第 9 章 贪婪策略

对于具有最优子结构性质和子问题重叠性质的组合优化问题，可以利用动态规划方法有效地加以计算。如果一个组合优化问题除了最优子结构特性外，还具有一些更深入的特性，则可以用更特殊的方法来提高算法的效率。本章就来讨论一种称为“贪婪选择”的方法。适合于运用贪婪选择方法的问题，除了具有最优子结构之外还要具有贪婪选择性质——自顶向下对每一个子问题计算到目前为止最优的部分解。这样，最终得到的解对于原问题而言是最优的。具有最优子结构性质和贪婪选择性质的问题，可以用贪婪选择策略设计出高效的算法。

9.1 活动选择问题

9.1.1 算法描述与分析

1. 问题的理解与描述

回顾在第 7 章 7.3.1 节中讨论过的相容活动问题。假定有 n 个需要使用同一个诸如教室这样的资源的活动 $S=\{a_1,a_2,\cdots,a_n\}$，每次只能有一个活动使用该资源。每一个活动有一个开始时间 $s[a_i]$，一个完成时间 $f[a_i]$，其中 $0\leqslant s[a_i]<f[a_i]<\infty$。若活动 a_i 被选取，则它在半闭半开区间 $[s[a_i],f[a_i])$ 内发生。活动 a_i 和 a_j 是**相容的**，如果区间 $[s[a_i],f[a_i])$ 和 $[s[a_j],f[a_j])$ 不相交（即如果 $s[a_i]\geqslant f[a_j]$ 或 $s[a_j]\geqslant f[a_i]$，a_i 和 a_j 相容）。活动选择问题是选取一个由相容活动构成的最大集合。例如，考虑下列活动集合 S，其中活动按完成时间的单调增加顺序排列：

i	1	2	3	4	5	6	7	8	9	10	11
$s[a_i]$	1	3	0	5	3	5	6	8	8	2	12
$f[a_i]$	4	5	6	7	8	9	10	11	12	13	14

不久就会看到把活动排序的好处。在此例子中，子集 $\{a_3,a_9,a_{11}\}$ 由相容活动组成。但它不是最大的，因为 $\{a_1,a_4,a_8,a_{11}\}$ 是更大的这样的子集。事实上，$\{a_1,a_4,a_8,a_{11}\}$ 是一个最大的相容活动的子集；另一个最大的子集是 $\{a_2,a_4,a_9,a_{11}\}$。这是一个组合优化问题：S 的任何一个子集都是一个可能解，相容活动构成的子集为可行解。每一个可行解都以其所含活动个数作为目标值。目的是找到一个目标值最大的可行解。

输入：按完成时间排好序的活动组 $S=\{a_1,a_2,\cdots,a_n\}$。

输出：表示一个最大的相容活动组的向量 $\{x_1,x_2,\cdots,x_n\}$，其中

$$x_i=\begin{cases}1 & \text{活动 } a_i \text{ 在此最大相容活动组中}\\ 0 & \text{否则}\end{cases},\quad i=1,2,\cdots,n$$

2. 最优子结构

由于所有可能解对应 2^n 个向量$\{x_1,x_2,\cdots,x_n\}$,$x_i\in\{0,1\}$,$i=1,2,\cdots,n$。所以用回溯算法的时间复杂度是 $O(2^n)$。为了提高算法效率,先来考察该问题是否具有最优子结构特性。为此,定义集合:

$$S_{ij}=\{a_k\in S: f[a_i]\leqslant s[a_k]<f[a_k]\leqslant s[a_j]\}$$

也就是说 S_{ij} 是由所有与 a_i 和 a_j 相容的活动构成,这些活动也与所有不迟于 a_i 完成而完成以及不先于 a_j 的开始才开始的活动相容。为表示整个问题,添加两个假想的活动 a_0 和 a_{n+1},并约定 $f[a_0]=0$ 及 $s[a_{n+1}]=\infty$。于是 $S=S_{0n+1}$,而 i 和 j 的取值范围为 $0\leqslant i$,$j\leqslant n+1$。

由于假定各活动按完成时间的单调增加顺序排列:

$$f[a_0]\leqslant f[a_1]\leqslant f[a_2]\leqslant\cdots\leqslant f[a_n]<f[a_{n+1}]$$

根据 S_{ij} 的定义,显然只要 $i\geqslant j$,就有 $S_{ij}=\varnothing$。于是,子问题空间是在 S_{ij},$0\leqslant i<j\leqslant n+1$ 中选取最大的相容活动子集。假定 S_{ij} 的一个解包含活动 a_k,所以 $f[a_i]\leqslant s[a_k]<f[a_k]\leqslant s[a_j]$。利用$a_k$ 产生两个子问题,S_{ik}(发生在 a_i 完成以后且完成于 a_k 发生之前的活动)和 S_{kj}(发生在 a_k 完成以后且完成于 a_j 发生之前的活动),每一个都构成 S_{ij} 中的活动的子集。设 A_{ij} 是 S_{ij} 的一个最优解,$a_k\in A_{ij}$。则 A_{ij} 中 S_{ik} 的解 A_{ik} 和 S_{kj} 的解 A_{kj} 必也是最优的。这是因为,若 S_{ik} 有一个比 A_{ik} 所包含的活动更多的解 A'_{ik},我们可以把 A_{ik} 从 A_{ij} 中切除并粘贴 A'_{ik},就会产生 S_{ij} 的另一个比 A_{ij} 包含更多活动的解。由于假定 A_{ij} 是一个最优解,这得出了一个矛盾。类似地,可以说明 A_{kj} 是 S_{kj} 的一个最优解。

现在利用此最优子结构来计算最优解的值。设 $c[i,j]$为 S_{ij} 的最大相容活动子集中的活动数,则:

$$c[i,j]=\begin{cases}0 & S_{ij}=\varnothing\\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & S_{ij}\neq\varnothing\end{cases}$$

到此为止,可以写一个基于上式的动态规划算法。

```
DYNAMIC-ACTIVITY-SELECTOR(S)
n (length[S]
for i←0 to n+1
    do for j←0 to n+1
           do c[i,j]←0
for l←2 to n
    do for i←1 to n−l+1
         do j←i+l−1
           for k←i+1 to j−1
              do if s[a_k]≥f[a_i] and f[a_k]≤s[a_j]
                    then t←c[i,k]+c[k,j]+1
                         if c[i,j]<t
                            then c[i,j]←t
```

算法 9-1 解决活动选择问题的动态规划算法

不难分析，该算法的时间复杂度是 $O(n^3)$。为了进一步提高算法效率，下面来考察该问题另一个有趣的特性。从 $c[i,j]$的定义可见，为了计算 $i<j$ 时的 $c[i,j]$，需要计算$\max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\}$，这里有两个层次的计算：首先，$k$ 从 $i+1$ 到 $j-1$ 的循环；其次，解两个子问题 $c[i,k]$和 $c[k,j]$。设想如果能一步就确定 k 的值并且能减少计算一个子问题，就一定能大大地提高算法的效率。定理 9-1 揭示了在活动按完成时间排好序的前提条件下，确实能在这两个方面改进算法。

3. 贪婪选择性

定理 9-1　考虑任一非空子问题 S_{ij}，并设 a_m 为 S_{ij} 中最早完成的一个活动：

$$f[a_m] = \min\{\, f[a_k] : a_k \in S_{ij} \}$$

则

(1) 活动 a_m 包含在 S_{ij} 的一个最大相容活动子集合中。

(2) 子问题 S_{im} 是空的，所以选择 a_m 将使得 S_{mj} 成为仅有的非空子问题。

为说明第一部分，假定 A_{ij} 是 S_{ij} 的一个最大相容活动子集合，并将 A_{ij} 中的活动按完成时间单调增加的顺序排列。设 a_k 是 A_{ij} 中的第一个活动。若 $a_k=a_m$，就完成了证明，因为已经说明 a_m 是 S_{ij} 的一个最大相容活动子集合的成员之一。若 $a_k\neq a_m$，构造子集合 $A'_{ij}=A_{ij}-\{a_k\}\cup\{a_m\}$。$A'_{ij}$ 中的活动是互不相交的，这是因为 a_k 是 A_{ij} 中第一个完成的活动，且 $f_m\leqslant f_k$。注意 A'_{ij} 和 A_{ij} 所含的活动数一样，我们知道 A'_{ij} 是 S_{ij} 的一个包含 a_m 的最大相容活动子集。

接下来说明第二部分。假定 S_{im} 非空，所以有活动 a_k 使得 $f[a_i]\leqslant s[a_k]<f[a_k]\leqslant s[a_m]<f[a_m]$，而$a_k$ 也在 S_{ij} 中，而且是一个比 a_m 完成得更早的活动，此与 a_m 的选择矛盾。我们推出 S_{im} 是空的。

4. 算法的伪代码描述

定理 9-1 说明如果“贪婪地”取 S_{ij} 中最先完成的活动 a_m，则第 1 个结论告诉我们该活动必存在于该问题的一个最优解中。第 2 个结论则确定了无须计算 $c[i,m]$，因为子问题 S_{im} 是空的。于是只需要递归地解决子问题 S_{mj}。这样，就可以自顶向下地对每一个活动确定是否含在一个最大相容集中。

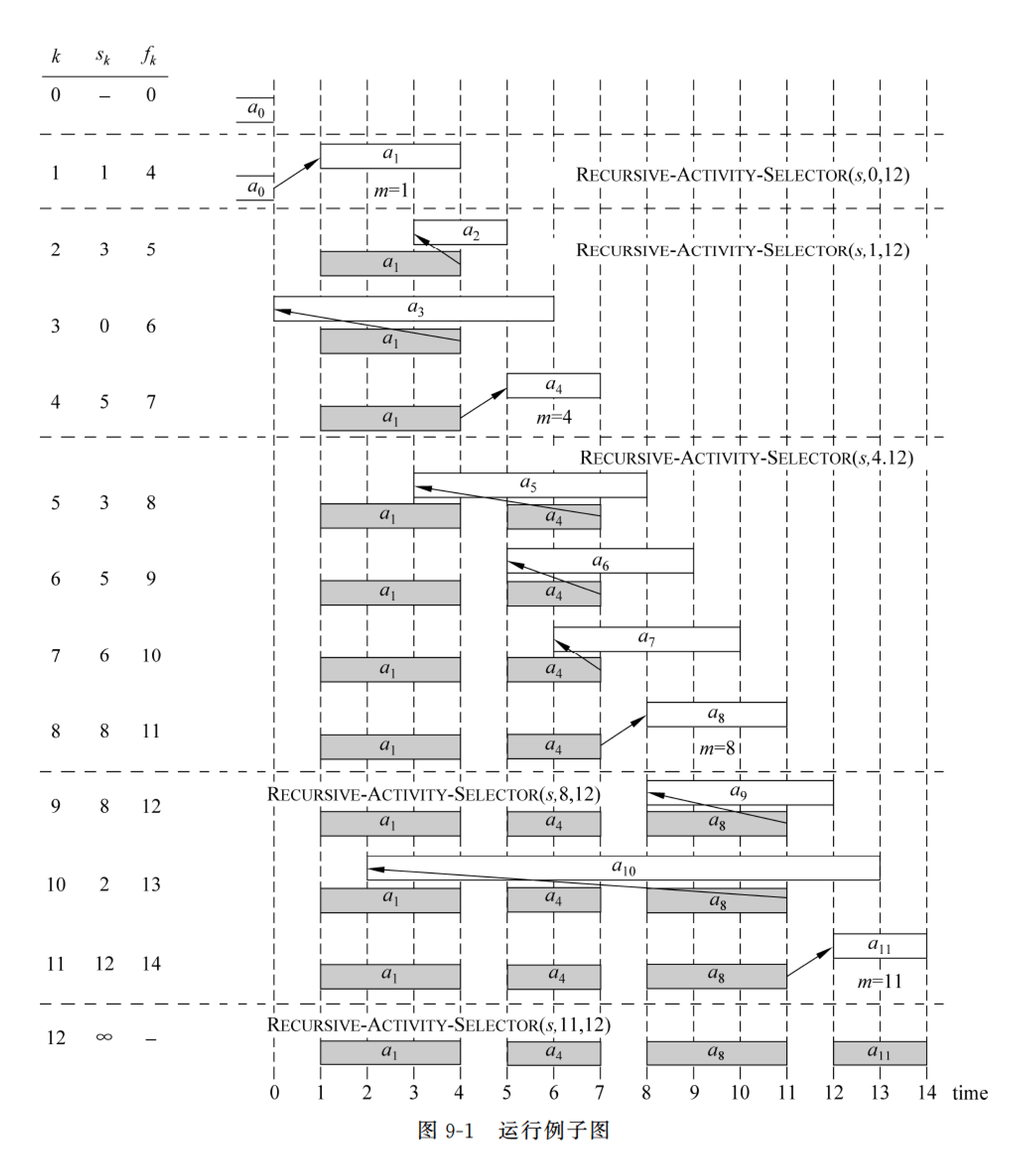
```
RECURSIVE-ACTIVITY-SELECTOR (S,i,j) ▷S={a1,a2,…,an}
while m<j 且 s[am]<f[ai]                  ▷求 Sij 中的第一个活动
    do xm←0
       m←m+1
if m<j
    then xm←1
         RECURSIVE-ACTIVITY-SELECTOR(S,m,j)
```

算法 9-2　解决活动选择问题的递归算法

算法运行于一个活动组的例子如图 9-1 所示。

图 9-1 中，RECURSIVE-ACTIVITY-SELECTOR 运行于前面给出的 11 个活动。每次递归



k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	–

图 9-1 运行例子图

调用所考虑的活动位于两条水平线之间。假想的活动 a_0 在时间 0 完成。并在首次调用 RECURSIVE-ACTIVITY-SELECTOR(s,f,0,12)中,活动 a_1 被选取。在每一次递归调用中,已经被选取的活动加了阴影,显示为白色的活动正被考察。若一个活动的开始时间发生在最近加入的活动完成之前(两者之间的箭头朝向左边),它被拒绝。否则的话(箭头朝上或朝右),它被选取。最后一次递归调用 RECURSIVE-ACTIVITY-SELECTOR(s,f,11,12),返回 $\varnothing$。选取的结果活动为$\{a_1,a_4,a_8,a_{11}\}$。

由于过程 RECURSIVE-ACTIVITY-SELECTOR 是“末尾递归”。将一个末尾递归过程转换为与之等价的迭代形式往往是很直接的。

```
GREEDY-ACTIVITY-SELECTOR(S)                    ▷ S={a1,a2,…,an}
n←length[S]
```

```
for i ← 1 to n
    do x_i ← 0
x_1 ← 1
i ← 1
for m ← 2 to n
    do if s[a_m] ≥ f[a_i]
        then x_m ← 1
             i ← m
return x
```

算法 9-3　解决活动选择问题的迭代算法

5. 算法的运行时间

显然，GREEDY-ACTIVITY-SELECTOR 的运行时间是 $\Theta(n)$，这比动态规划算法的 $\Theta(n^3)$ 好得多。由此可见，对于具有最优子结构特性的组合优化问题，若还具有贪婪选择特性——总是选择目前的最优部分解，且该部分解包含于一个全局最优解中——则可进一步提高算法效率。

9.1.2　程序实现

1. 实现要点

要将一个算法的伪代码过程实现为一个程序过程（函数或方法），需要考虑过程与外部通信的机制——参数与返回值，前者对应于问题的输入，后者对应于问题的输出。程序过程中运行时需要设置的数据变量以及如何向计算机解释伪代码中高度抽象的表达，或如何利用语言所具有的特点来实现伪代码的特殊表达。在把算法 9-1 的伪代码过程实现为程序过程时，也遵循这样的思路。

【参数与返回值】　程序过程和算法伪代码一样，参数包括分别表示活动的起始时间数组 s、完成时间数组 f、当前活动组起始编号 i 和终点编号 j。它们的类型都应该是整型的。若解向量作为全局变量，则该过程无返回值；否则需要返回解向量（一个整型的一维数组）。

【数据设置】　可以和算法 9-1 那样，将解向量 x 设置为全局变量。其实，它可以通过参数传递给该函数。此外还需要设置一个整型变量 m 作为循环控制变量。

【关键代码】　伪代码十分简单，可以直接转换为程序代码。

2. 程序实现

用 C 语言可以很轻松地实现算法 9-3。

```
typedef struct{
    double s;
    double f;
}Active;
int activeComp(Active *a,Active *b){
    if(a->f>b->f)
```

```
7           return 1;
8       if(a->f<b->f)
9           return -1;
10      return 0;
11 }
12 int *greedyActivitySelector(Active *a,int n){
13      int i,m,*x=(int *)calloc(n,sizeof(int));
14      assert(x);
15      qsort(a,n,sizeof(Active),activeComp);
16      for(x[0]=1,i=0,m=1;m<n;m++)
17          if(a[m].s>=a[i].f){
18              x[m]=1;
19              i=m;
20          }
21      return x;
22 }
```

程序 9-1 实现算法 9-3 的 C 代码

对程序 9-1 的说明如下。

(1) 第 1~4 行将活动定义成具有两个属性 s、f 的结构体类型 Active。为适应各种应用,属性 s 和 f 的类型均为 **double** 类型。

(2) 第 5~11 行定义的函数 activeComp 对由两个参数指针 a、b 指引的活动比较完成时间 f 的大小。它是对活动组进行排序的比较规则。

(3) 第 12~22 行定义的函数 greedyActivitySelector 实现算法 9-2。该函数的第 1 个参数 a 表示活动数组,第 2 个参数 n 表示 a 中的活动个数。与算法过程一样,函数返回表示 a 中的一个最大相容子集的向量 x。由于 a 中活动未必按完成时间排好序,所以第 15 行调用库函数 qsort,传递 activeComp,对 a 按完成时间升序排序。函数体内其余代码与算法 9-1 的伪代码十分接近,读者可对比阅读。

9.1.3 贪婪算法与动态规划

值得注意的是,并非所有具有最优子结构特性的组合优化问题都具有贪婪选择特性。考察下列两个问题。

0-1 背包问题:有 n 件物品,第 i 件物品价值为 v_i,质量为 w_i,其中 v_i 和 w_i 是整数。希望尽可能地用背包带走质量为 w 的值钱东西(对于每件物品,要么留下,要么整件带走),其中C 是整数。问题是应该带走哪些东西?即在 $\sum_{i=1}^{n} x_i w_i \leqslant C, x_i \in \{0,1\}, i=1,\cdots,n$ 的限制下,最大化 $\sum_{i=1}^{n} x_i v_i$。

部分背包问题:有 n 件物品,第 i 件物品价值为 v_i,质量为 w_i,其中 v_i 和 w_i 是整数。希望尽可能地用背包带走质量为 w 的值钱东西(对于每件物品,可以带走一部分),其中 C

是整数。问题是应该带走哪些东西？即在 $\sum_{i=1}^{n} x_i w_i \leqslant C, x_i \in [0,1], i=1,\cdots,n$ 的限制下，最大化 $\sum_{i=1}^{n} x_i v_i$。

这两个背包问题都呈现出最优子结构性质。对 0-1 背包问题而言，其最优子结构特性已经在第 3 章中阐述。对于部分背包问题而言，若从最优装载中移除质量为 w 的物品 j，剩下的必是要带走的至多质量为 $C-w$ 最有价值的原 $n-1$ 种物品和质量为 w_j-w 的物品 j。

虽然两个问题很相似，若对每一种物品计算单位价值 v_i/w_i。贪婪策略是依次取走尽可能多的单位价值最高的物品。部分背包问题可以用贪婪策略加以解决，而 0-1 背包问题却不行，实例见图 9-2。所以，在用贪婪策略解决一个组合优化问题之前需要仔细考察它的最优子结构特性和贪婪选择特性。

图 9-2(a)中，选择 3 个物品的一个子集，质量不超过 50。图 9-2(b)中，0-1 背包问题的最优子集包含物品 2 和物品 3。任一含有物品 1 的解都不是最优的，尽管物品 1 具有最大的每磅价值。图 9-2(c)中，对部分背包问题，依次取最大的单位价值物品导致最优解。

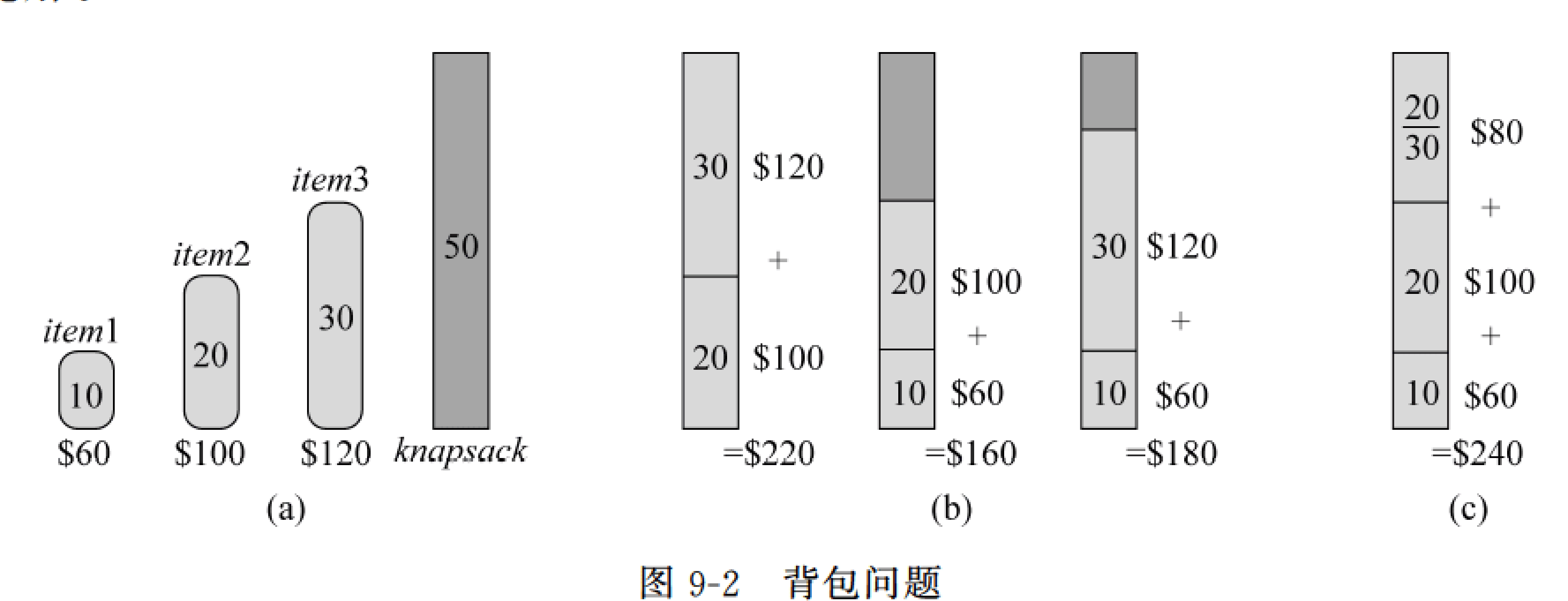


图 9-2　背包问题

一般说来，如果一个问题具有最优子结构就可以考虑用动态规划，只有当问题同时还具有贪婪选择性时，才能利用贪婪策略提高算法效率。

9.1.4　应用——海岸雷达

Radar Installations

Assume the coasting is an infinite straight line. Land is in one side of coasting, sea in the other. Each small island is a point locating in the sea side. And any radar installation, locating on the coasting, can only cover d distance, so an island in the sea can be covered by a radius installation, if the distance between them is at most d.

We use Cartesian coordinate system, defining the coasting is the x-axis. The sea side is above x-axis, and the land side below. Given the position of each island in the sea, and given the distance of the coverage of the radar installation, your task is to write a program to find the minimal number of radar installations to cover all the islands. Note that the position of an island is represented by its x-y coordinates.

Input

The input consists of several test cases. The first line of each case contains two integers n ($1 \leqslant n \leqslant 1000$) and d, where n is the number of islands in the sea and d is the distance of coverage of the radar installation. This is followed by n lines each containing two integers representing the coordinate of the position of each island. Then a blank line follows to separate the cases.

The input is terminated by a line containing pair of zeros.

Output

For each test case output one line consisting of the test case number followed by the minimal number of radar installations needed. "−1" installation means no solution for that case.

Figare 9-3 is a sample input of radar Installations.

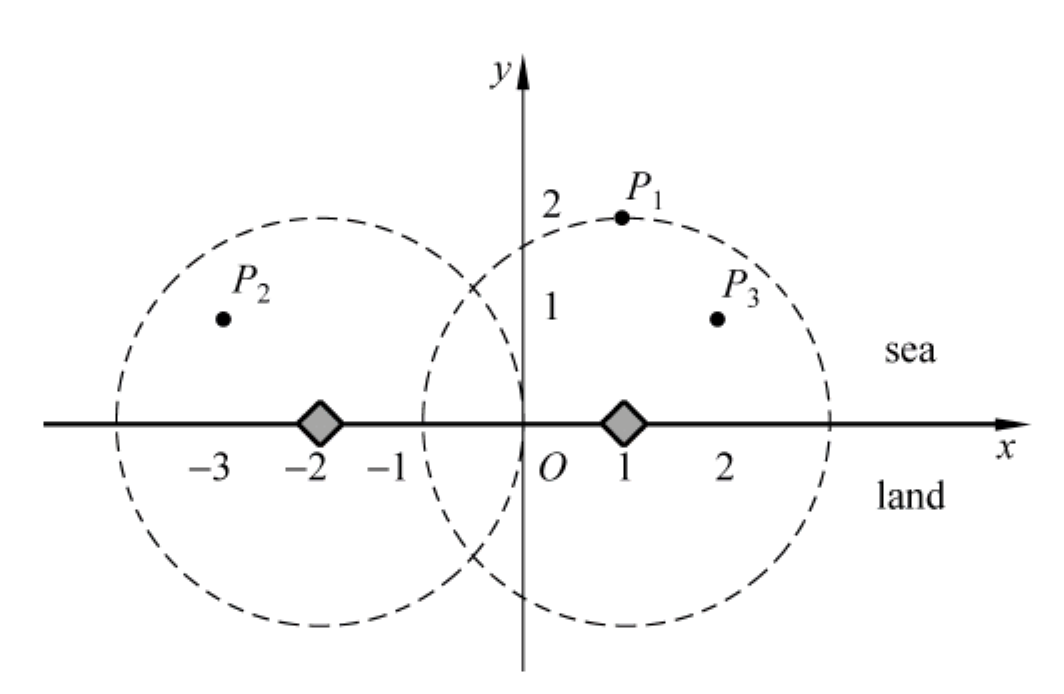


图 9-3　A Sample Input of Radar Installations

Sample Input

```
3 2
1 2

-3 1
2 1

1 2
0 2


0 0
```

Sample Output

```
Case 1: 2
Case 2: 1
```

1. 问题描述与分析

设海岸线为直角坐标系中的 x 轴。x 轴上方表示海,下方表示陆地。在海岸线上安装扫描半径为 d 的雷达监视位于海中坐标为(x_i, y_i), $i=1,2,\cdots,n$ 的 n 个岛屿。对一个位于

(x,y)的岛屿来说，能监视它的雷达只能位于作为海岸线的 x 轴上的区间$[x-\sqrt{d^2-y^2},x+\sqrt{d^2-y^2}]$内。于是，两个这样区间相交意味着两个岛屿可被同一台雷达所监视。区间不相交，则意味着两个岛屿不可能被同一台雷达监视。因此，最少的雷达安装数，就是由 n 个岛屿的位置所决定的 n 个区间最大相容(使用活动选择问题的语言，此处所说的相容指的是两个区间不相交)数。这样，本问题就可以转换成活动选择问题了。把问题形式化为如下。

输入：表示 n 个岛屿位置坐标的两个数组 x、y。

输出：区间集合$\{[x_i-\sqrt{d^2-y_i^2},x_i+\sqrt{d^2-y_i^2}]\mid i=1,2,\cdots,n\}$的最大不相交区间构成的子集所含区间个数。

2. 算法描述

利用活动选择问题算法过程 GREEDY-ACTIVITY-SELECTOR 可将解决此问题的算法伪代码表示为如下。

```
RADAR-INSTALLATION(x,y)
n←length[x]
for i←1 to n
    do l_i←x_i − sqrt(d^2 − y_i^2)
       r_i←x_i + sqrt(d^2 − y_i^2)
按 r 的升序对 l 和 r 排序
z←GREEDY-ACTIVITY-SELECTOR (l,r)
return z 中值为 1 的元素个数
```

算法 9-4　解决 Radar Installations 问题的伪代码过程

3. 程序实现

```
int main(){
    FILE *f1=fopen("chap09/Radar Installations/inputdata.txt","r"),
         *f2=fopen("chap09/Radar Installations/outputdata.txt","w");
    int n,d,count=0;
    assert(f1&&f2);
    fscanf(f1,"%d%d",&n,&d);
    while(n&&d){
        int i,*s,min=0;
        double x,y,z;
        Active *a=(Active*)malloc(n*sizeof(Active));
        for(i=0;i<n;i++){
            fscanf(f1,"%lf%lf",&x,&y);
            z=sqrt(d*d-y*y);                    /*计算 √(d²-y²) */
            a[i].s=x-z;
            a[i].f=x+z;
        }
        s=greedyActivitySelector(a,n);
        for(i=0;i<n;i++)
```

```
19            min+=s[i];
20        fprintf(f2,"Case %d:%d\n",++count,min);
21        free(a);free(s);
22        fscanf(f1,"%d%d",&n,&d);
23    }
24    fclose(f1);fclose(f2);
25    return 0;
26 }
```

程序 9-2 解决 Radar Installations 问题的 C 程序

对程序 9-2 的说明如下。

(1) 第 2 行和第 3 行设置输入、输出文件 f1 和 f2。

(2) 第 6～23 行的 **while** 循环处理输入文件中的每个案例。其中第 6 行读取第一个案例的岛屿数 n 和雷达扫描半径 d,第 22 行在处理完每个案例数据后读取下一个案例的 n 和 d。

(3) 循环体中,第 10～16 行负责读取案例中的各岛屿坐标,并计算该岛屿对应的雷达安装区间,记录为数组 a 中一个元素的 s、f 属性。第 17 行调用程序 9-1 中定义的函数 greedyActivitySelector,返回向量 s。第 18 行和第 19 行计算 s 中为 1 的分量数作为变量 min 的值。第 20 行将 min 的值写入输出文件 f2 中。

9.2 Huffman 编码

9.2.1 算法描述与分析

1. 问题的理解与描述

Huffman 编码是一种广泛应用且非常有效的数据压缩技术;根据被压缩的数据特征,通常可以节省 20%～90%的空间。把数据视为一连串的字符,Huffman 的贪婪算法利用一张记录各字符发生频率的表格建立一个将每个字符表示为二进制串的最优方式。

假定希望压缩存储一个具有 100 000 个字符的数据文件。我们观察到该文件中的字符发生的频率由图 9-4 给出,即仅有 6 个不同的字符,且字符 a 发生了 45 000 次。

	a	**b**	**c**	**d**	**e**	**f**
频率(单位为千)	45	13	12	16	9	5
定长编码字	000	001	010	011	100	101
变长编码字	0	101	100	111	1101	1100

图 9-4 一个字符编码问题

有很多种方式用来表示这样的信息文件。我们考虑设计一种二进制编码(简称为**编码**),其中的每个字符表示为唯一的二进制串。若使用**定长编码**,需要 3 个比特来表示 6 个字符:a=000,b=001,…,f=101。这个方法对整个文件编码需要 300 000 个比特。另一方

面，**变长编码**通过赋予高频字符短编码字而低频字符长编码字可以比定长编码做得好得多。图 9-4 展示了这样的一个编码；其中用一个比特的串 0 表示 a，4 比特的串 1100 表示 f。这一编码需要

$$(45\times 1+13\times 3+12\times 3+16\times 3+9\times 4+5\times 4)\times 1000=224\,000 \text{ 比特}$$

来表示文件，几乎节省 25%。事实上将看到这是对此文件的最优字符编码。

上例中，无论是定长编码还是变长编码，都具有共同的特性：没有一个编码字成为另一个编码字的前缀，这样的编码称为**前缀码**。

对前缀码来说编码是很简单的，只要把文件中每个字符的编码字连接起来就行了。例如，用图 9-4 中的变长前缀码，对 3 个字符的文件 abc 编码为 0・101・100＝0101100，其中用"・"表示连接。

前缀码因为解码简便而受到欢迎。由于没有哪个编码字是另一个编码字的前缀，一个已编码文件的第一个编码字是确定的。可以直接识别首个编码字，将其转换为原字符，然后对剩下的已编码文件重复此过程。在例子中，串 001011101 分解成 0・0・101・1101，解码为 aabe。为确定第一个编码字，利用一棵以给定的字符为树叶的二叉树的表示方法。把一个字符的二进制编码字解释为从根起到该字符的路径，其中 0 意为"行进到左孩子"，而 1 意为"行进到右孩子"。图 9-5 展示了两棵表示例子的编码树。把这样的用来表示字符集前缀码的二叉树称为一棵**编码树**。注意，编码树中除了对应于字符的每一片叶子有其频数，每一个内点也有其"频数"：定义为它的两个孩子的频数之和。

图 9-5 所示为对应于图 9-4 的编码方案的树。每一片叶子标注了一个字符及其发生的频数。每一个内点标示了子树中所含叶子的频数之和。图 9-5(a)对应于定长编码的树 $a=000,\cdots,f=101$。图 9-5(b)对应于最优前缀码的树 $a=0,b=101,\cdots,f=1100$。

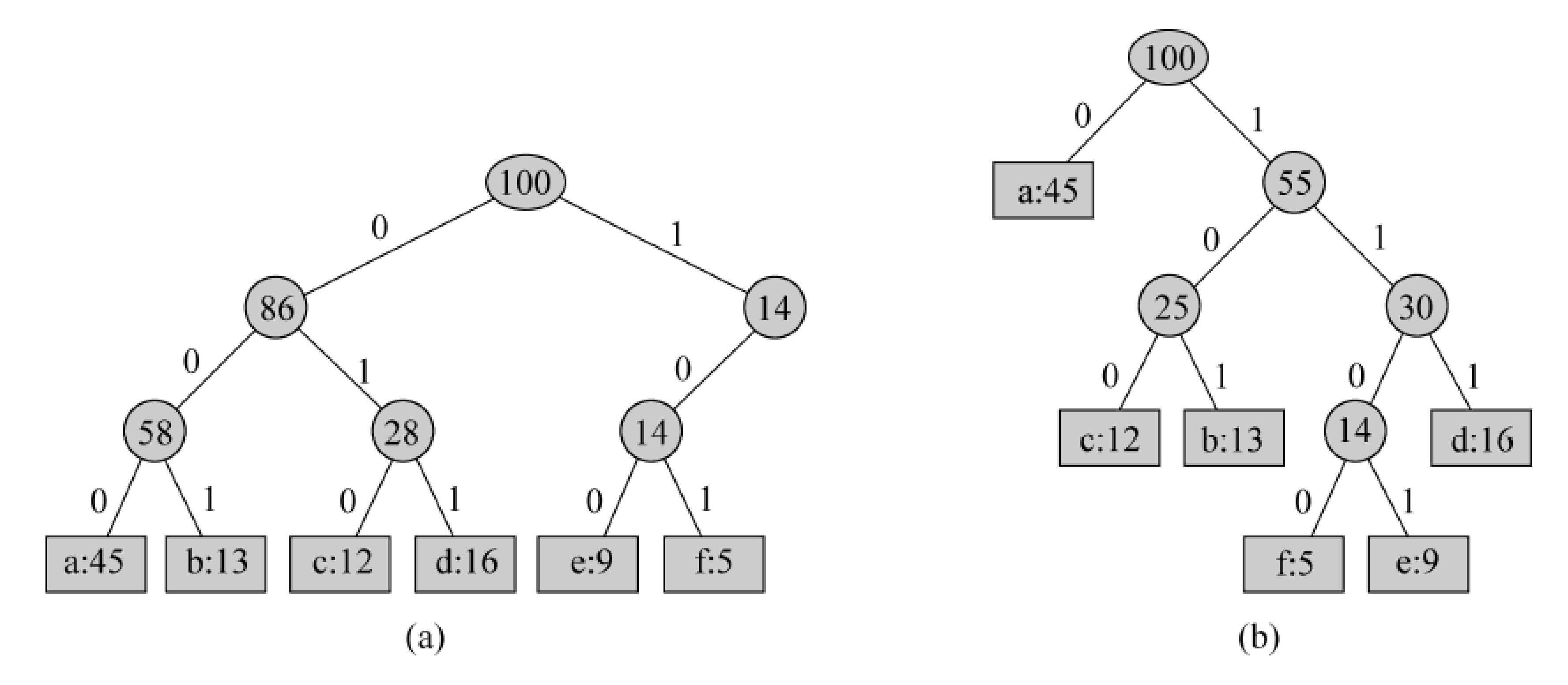


图 9-5　对应于图 9-4 的编码方案的树

给定一棵对应于前缀码的编码树 T，计算编码一个文件所需的比特数——文件长度——是很简单的。对字母集 C 中的每一个字符 c，$f(c)$表示 c 在文件中发生的频数，$d_T(c)$表示c 的叶子在树中的深度。注意 $d_T(c)$还表示字符 c 的编码字长度。于是对该文件编码所需的比特长度为

$$B(T)=\sum_{c\in C} f(c)d_T(c) \tag{9-1}$$

把它定义为树 T 的**代价**。

同一个文件的不同的前缀码编码方案具有不同的文件长度。对应于最短文件长度的前缀码编码方案,称为该文件一个**最优编码**。一个文件的最优编码所对应的编码树必为一棵满二叉树[①],在树中每一个内点都有两个孩子。这是因为如果编码树不满,总可以得到一棵代价更小的编码树。例子中的定长编码不是最优的,因为图 9-5(a)中所示的对应编码树不是满的:有始于 10…的编码字,但没有始于 11…的编码字。将由叶子 e、f 构成的子树移到其父结点处将得到一棵代价更小的编码树(当然对应的编码不再是等长码了)。若 C 是字母集且所有字符的频数都是正的,则对 C 的一个最优前缀码而言,它所对应的满二叉树具有 $|C|$ 片树叶,每片叶子对应字母集中的一个字符,且恰有 $|C|-1$ 个内点。

把问题形式化为如下。

输入:字符集 C 及频数 $f(c)$,$c\in C$。

输出:以 C 中字符为叶子的满二叉树 T,使得代价 $B(T)=\sum_{c\in C}f(c)d_T(c)$ 最小。

2. 最优子结构

显然这是一个组合优化问题。由于一共有 $\frac{1}{n+1}\binom{2n}{n}$ 棵具有 n 片叶子且互不同构的满二叉树[②],所以可构造以 C 中字符为叶子的 $\frac{1}{|C|+1}\binom{2|C|}{|C|}$ 棵前缀码树,每棵树 T 对应一个代价 $B(T)$ 作为目标值,目的是找一棵目标值最小的前缀码树。该优化问题具有最优子结构特性:设 T 为 C 的一棵最优前缀码树,T 的两个孩子分别为 T_1 和 T_2。设 C_1 和 C_2 分别是 T_1 和 T_2 中叶子所对应的字符集。显然,$C_1\cup C_2=C$,$C_1\cap C_2=\varnothing$。则 T_1 和 T_2 分别为 C_1 和 C_2 的最优前缀码树。若不然,假设对 C_1 而言,有一棵优于 T_1 的前缀码树 T_1',$B(T_1')<B(T_1)$。这样,以 T_1' 和 T_2 为孩子的 C 的前缀码树 T',必有 $B(T')<B(T)$。这与 T 是 C 的一棵最优前缀码树矛盾。

3. 贪婪选择性

按如下的贪婪策略来构成一棵前缀码树:每一次从当前的满二叉树中(初始时,每一个字符构成一棵平凡的二叉树,共有 n 棵)选择两棵频数最小的构造成一棵新的满二叉树。由于每做一次这样的操作,就会减少一棵树,所以 $n-1$ 次操作后将构成唯一的一棵满二叉树。定理 9-2 告诉我们,本问题具有贪婪选择性质。

定理 9-2 设 C 为一个字母表,其中的每一个字符 $c\in C$ 具有频数 $f[c]$。设 x 和 y 是 C 中的两个频数最小的字符。则

(1) 存在 C 的一棵最优前缀码树 T,在此编码树中 x 和 y 是兄弟叶子。

(2) 设 C' 是在 C 中移除字符 x、y 并加入新的字符 z 的字母表,所以 $C'=C-\{x,y\}\cup\{z\}$;定义 C' 中的 f 除了 $f[z]=f[x]+f[y]$ 外,其余的和在 C 中的一样。设 T' 为任一棵表示 C' 的一个最优前缀码的树。则在树 T' 中将叶子 z 替换为以 x 为左孩子、y 为右孩子的一个内

① 本书所说的满二叉树,指的是树中每个内点均有两个孩子的二叉树。

② 见第 3 章中的脚注 1。

点得到的树 T 为表示字母表 C 的一个最优前缀码的树。

本定理结论(1)证明的思想是取一棵表示任意最优前缀码的树 T，修改此树使之成为表示另一个最优前缀码的树，在这棵新树中 x 和 y 为两片具有最大深度的兄弟叶子。

设 a 和 b 为 T 中具有最大深度的两片兄弟叶子。由于 $f[x]$和 $f[y]$是两个最小的频数，而 $f[a]$和 $f[b]$是两个任意的频数，所以有 $f[x]\leqslant f[a]$且 $f[y]\leqslant f[b]$。分别交换 a 和 x、b 和 y 产生树 T'。由于每一个交换并不增加代价，所以 T'也是最优的，如图 9-6 所示。

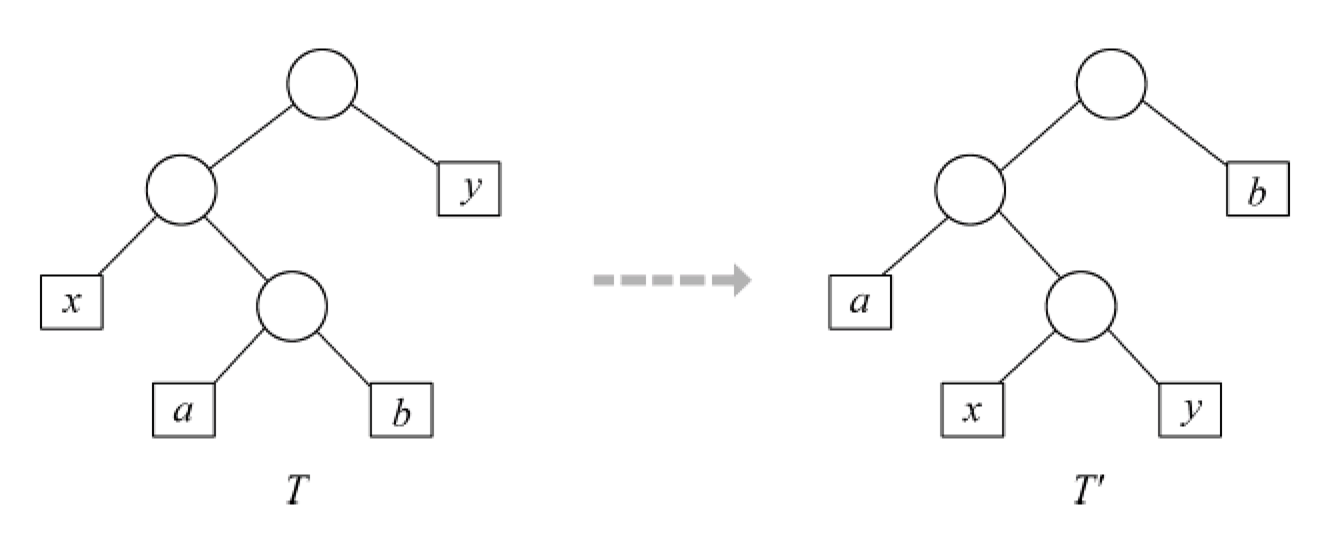


图 9-6　定理证明示例

而对定理中的结论(2)而言，首先从图 9-7 中可见 $B(T)$和 $B(T')$之间有如下关系：

$$B(T) = B(T') + f[x] + f[y]$$

或，等价地

$$B(T') = B(T) - f[x] - f[y]$$

图 9-7 中，T'是将 C 的一棵编码树 T 中以 x、y 为孩子的内点替换成新的叶子 z，且 $f[z]=f[x]+f[y]$，构成C'的一棵编码树。由于 T 中仅有 x、y 的深度比 T'中 z 的深度多一层，因此有 $B(T')=B(T)-f[x]-f[y]$。

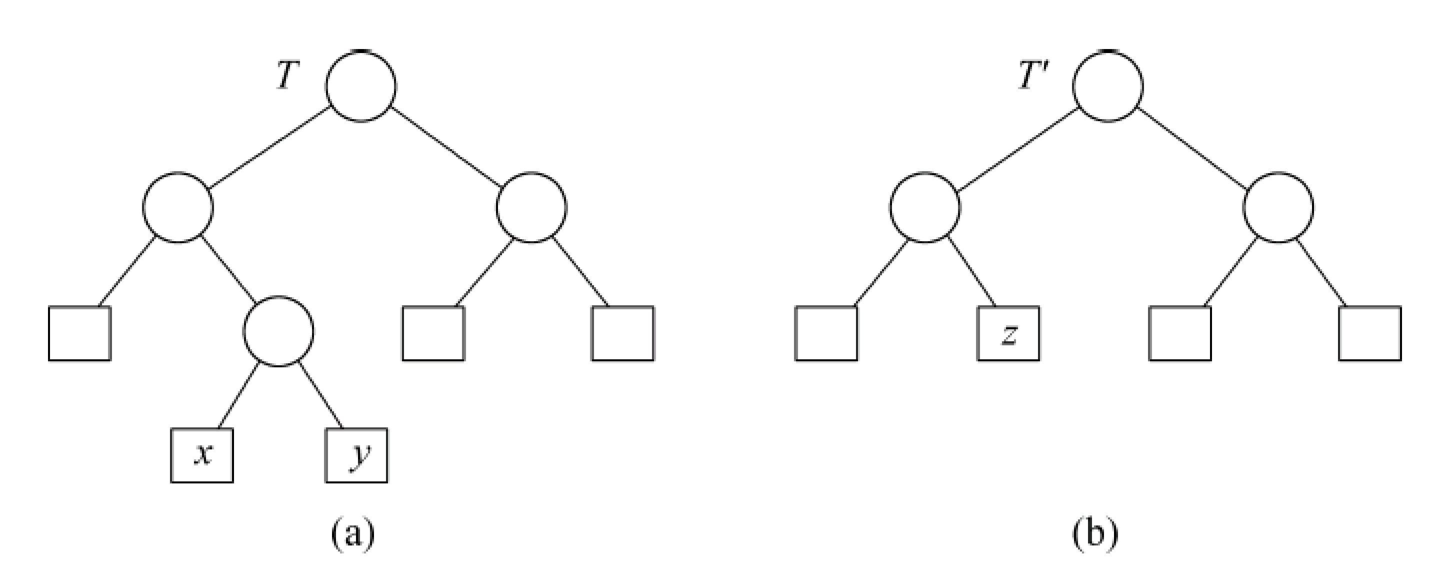


图 9-7　编码树

用反证法来说明定理的结论(2)。假定 T 并不表示 C 的一个最优前缀码。则存在最优编码树 T''满足 $B(T'')<B(T)$。不失一般性(根据(1))，T''中 x 作为 y 的兄弟。设 T'''为将树 T''中的 x 和 y 的共同父母替换成叶子 z，其频数为 $f[z]=f[x]+f[y]$。于是：

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T')$$

此与 T'表示 C'的一个最优前缀码矛盾。因此，T 必表示字母表 C 的一个最优前缀码编码树。

4. 算法的伪代码描述

利用本问题的最优子结构性质和贪婪选择性质，Huffman 发明了一个创建称为

Huffman 编码的最优前缀码的贪婪算法。

```
HUFFMAN(C)
n←|C|
Q←C
for i←1 to n−1
    do 分配新的结点 z
        left[z]←x←DEQUEUE(Q)
        right[z]←y←DEQUEUE(Q)
        parent[x]←parent[y]←z
        f[z]←f[x]+f[y]
        ENQUEUE(Q,z)
return DEQUEUE(Q)                    ▷返回树的根
```

算法 9-5 解决 Huffman 编码问题的算法

过程 HUFFMAN 中的 Q 是一个用来存储子树的优先队列,每棵树的根结点记录下树中所含叶子结点的频数之和作为优先级。对于上述的例子而言,Huffman 算法的运行如图 9-8 所示。因为字母表中有 6 个字符,队列初始时大小为 6,建立树需要 5 个合并步骤。最终的树表示该最优编码。一个字符的编码字是从根到该字符路径各条边上的标注的序列。

图 9-8 所示为对图 9-3 中给出的频数 Huffman 算法执行的步骤。每一部分展示了按频数递增排序的队列内容。每一步合并两棵频数最低的树。内点表示为圆圈,其中包含了两个孩子的频数之和。连接内点与其孩子的边,若连接的是左孩子标示为 0,若连接的是右孩子标示为 1。一个字母的编码字是从根到表示该字母叶子路径上的边的标识序列。图 9-8(a)为最初的 $n=6$ 个点的集合,每个点表示一个字母。图 9-8(b)～图 9-8(e)为中间步骤,图 9-8(f)为

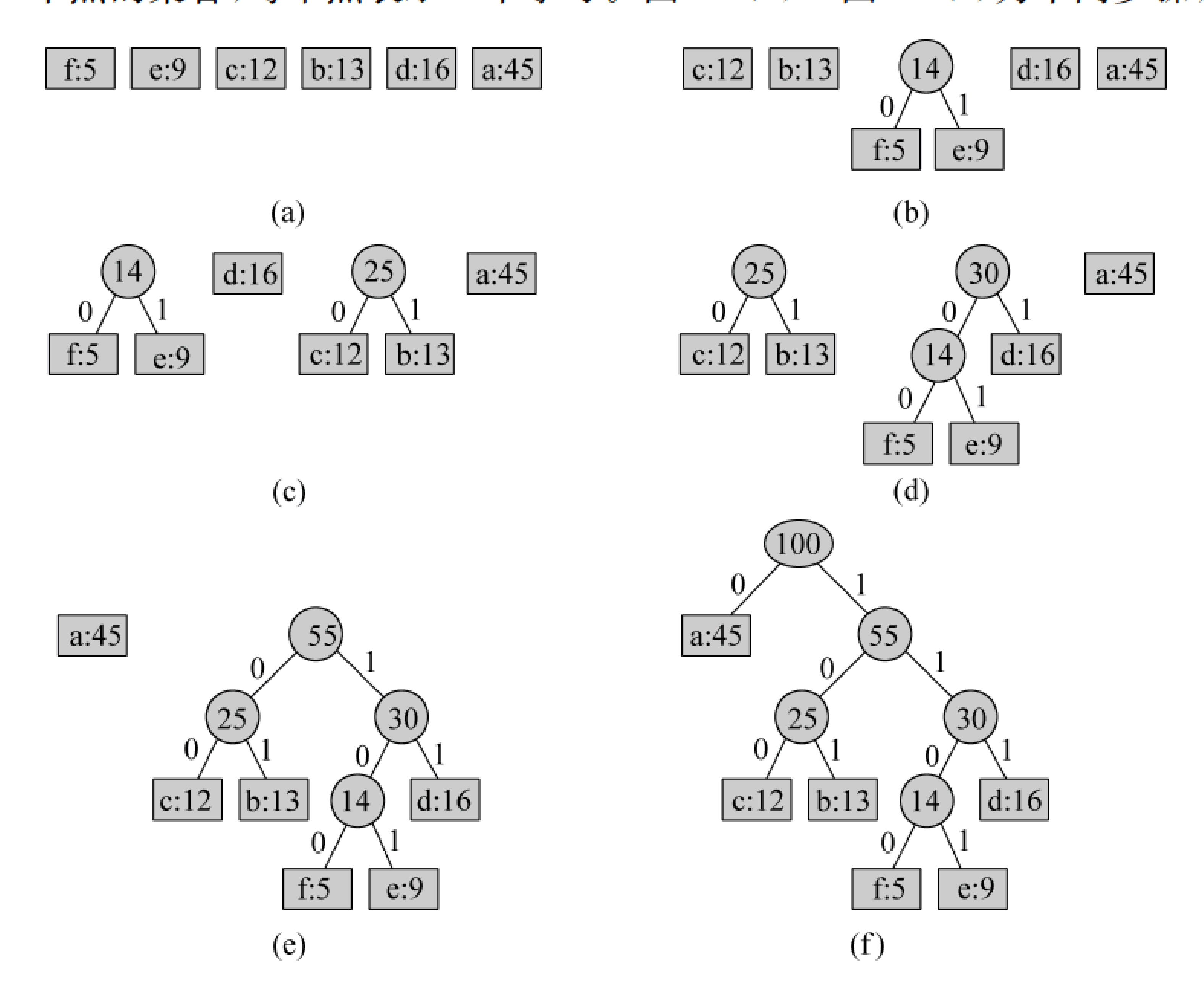


图 9-8 Huffman 算法的执行步骤

最终的树。

5. 算法的运行时间

根据第 2 章中对利用堆实现的优先队列的分析，第 2 行创建最小优先队列 Q 耗时 $\Theta(n)$①，第 5 行和第 6 行从优先队列 Q 中卸载队首各耗时 $\Theta(\lg n)$，第 8 行中将新的树插入到队列中耗时$\Theta(\lg n)$，第3～8 行的 **for** 循环重复 $\Theta(n)$次，所以过程 HUFFMAN 的总耗时为 $\Theta(n\lg n)$。

9.2.2　应用——*R* 叉 Huffman 树

Variable Radix Huffman Encoding

Description

Huffman encoding is a method of developing an optimal encoding of the symbols in a source alphabet using symbols from a target alphabet when the frequencies of each of the symbols in the source alphabet are known. Optimal means the average length of an encoded message will be minimized. In this problem you are to determine an encoding of the first N uppercase letters (the source alphabet, S_1 through S_N, with frequencies f_1 through f_N) into the first R decimal digits (the target alphabet, T_1 through T_R).

Consider determining the encoding when $R=2$. Encoding proceeds in several passes. In each pass the two source symbols with the lowest frequencies, say S_1 and S_2, are grouped to form a new "combination letter" whose frequency is the sum of f_1 and f_2. If there is a tie for the lowest or second lowest frequency, the letter occurring earlier in the alphabet is selected. After some number of passes only two letters remain to be combined. The letters combined in each pass are assigned one of the symbols from the target alphabet. The letter with the lower frequency is assigned the code 0, and the other letter is assigned the code 1. (If each letter in a combined group has the same frequency, then 0 is assigned to the one earliest in the alphabet. For the purpose of comparisons, the value of a "combination letter" is the value of the earliest letter in the combination.) The final code sequence for a source symbol is formed by concatenating the target alphabet symbols assigned as each combination letter using the source symbol is formed. The target symbols are concatenated in the reverse order that they are assigned so that the first symbol in the final code sequence is the last target symbol assigned to a combination letter. The two illustrations below demonstrate the process for $R=2$.

Symbol	Frequency		Symbol	Frequency
A	5		A	7
B	7		B	7
C	8		C	7

① 见本书第 2 章 2.4～2.6 节。

D 15	D 7
Pass 1: A and B grouped	Pass 1: A and B grouped
Pass 2: {A,B} and C grouped	Pass 2: C and D grouped
Pass 3: {A,B,C} and D grouped	Pass 3: {A,B} and {C,D} grouped
Resulting codes: A=110,B=111,C=10, D=0	Resulting codes: A=00,B=01, C=10,D=11
Avg. length=(3 * 5+3 * 7+2 * 8+1 * 15)/35=1.91	Avg. length=(2 * 7+2 * 7+2 * 7+2 * 7)/28=2.00

When R is larger than 2,R symbols are grouped in each pass. Since each pass effectively replaces R letters or combination letters by 1 combination letter, and the last pass must combine R letters or combination letters, the source alphabet must contain $k*(R-1)+R$ letters, for some integer k. Since N may not be this large, an appropriate number of fictitious letters with zero frequencies must be included. These fictitious letters are not to be included in the output. In making comparisons, the fictitious letters are later than any of the letters in the alphabet.

Now the basic process of determining the Huffman encoding is the same as for the $R=2$ case. In each pass, the R letters with the lowest frequencies are grouped, forming a new combination letter with a frequency equal to the sum of the letters included in the group. The letters that were grouped are assigned the target alphabet symbols 0 through $R-1$. 0 is assigned to the letter in the combination with the lowest frequency, 1 to the next lowest frequency, and so forth. If several of the letters in the group have the same frequency, the one earliest in the alphabet is assigned the smaller target symbol, and so forth. The illustration below demonstrates the process for $R=3$.

Symbol	Frequency
A	5
B	7
C	8
D	15

Pass 1: ? (fictitious symbol), A and B are grouped
Pass 2: {?,A,B}, C and D are grouped
Resulting codes: A=11,B=12,C=0,D=2
Avg. length=(2 * 5+2 * 7+1 * 8+1 * 15)/35=1.34

Input

The input will contain one or more data sets, one per line. Each data set consists of an integer value for R (between 2 and 10), an integer value for N (between 2 and 26), and the integer frequencies f_1 through f_N, each of which is between 1 and 999. The end of data for the entire input is the number 0 for R; it is not considered to be a separate data set.

Output

For each data set, display its number (numbering is sequential starting with 1) and the

average target symbol length (rounded to two decimal places) on one line. Then display the N letters of the source alphabet and the corresponding Huffman codes, one letter and code per line. The examples below illustrate the required output format.

Sample Input

```
2 5 5 10 20 25 40
2 5 4 2 2 1 1
3 7 20 5 8 5 12 6 9
4 6 10 23 18 25 9 12
0
```

Sample Output

```
Set 1; average length 2.10
    A: 1100
    B: 1101
    C: 111
    D: 10
    E: 0

Set 2; average length 2.20
    A: 11
    B: 00
    C: 01
    D: 100
    E: 101

Set 3; average length 1.69
    A: 1
    B: 00
    C: 20
    D: 01
    E: 22
    F: 02
    G: 21

Set 4; average length 1.32
    A: 32
    B: 1
    C: 0
    D: 2
    E: 31
    F: 33
```

1. 问题描述与分析

给定 N 个字符 S_1、…、S_N 及其在信息中出现的频数 f_1、…、f_N，要求以此计算信息按该

字符集 C 的最优编码的存储量。问题的形式化描述如下。

输入:字符集 $C=\{S_1,\cdots,S_N\}$及每个字符的频数 $f=\{f_1,\cdots,f_N\}$,正整数 R。

输出:C 的最优 R 进制编码下字符的平均存储量,C 中各字符的编码。

解决本问题的想法之一是将 Huffman 算法扩展到 R- 叉树,构造一棵 R 叉 Huffman 树 T,并计算出 T 的代价 $B(T)=\sum_{i=1}^{N}d_T(S_i)f_i$,进而计算各字符的平均存储量$\dfrac{\sum_{i=1}^{N}d_T(S_i)f_i}{\sum_{i=1}^{N}f_i}$。构造$R$ 叉 Huffman 树的方法是:初始时在 N 个字符中选取R 个频数最小的字符,并按频数从小到大(对频数相等的字符按字符表顺序)依次作为一棵 R 叉树的R 个孩子。将这棵树视为一个组合字符,其频数设置为R个孩子的频数之和,追加到字符集中。对新的字符集(字符个数比刚才少了 $R-1$),用同样的策略构成新的组合字符。若干次这样的操作,就能得到一棵 R 叉 Huffman 树。

要对 N 个字符构造R 叉编码树,即构造一棵有 N 片叶子的满R 叉树,可能会遇到树叶不够的问题。例如,题面中字符集为{A,B,C,D},仅有 $N=4$ 个字符,对 $R=3$,最小规模的满 3-叉树有 5 片叶子,就遇到了这一问题。为解决这一问题,题面给出了一个方法:若没有整数 k,使得 $k(R-1)+R=N$,取 $k=\min\{k|k\in\mathbf{N}$ 且 $k(R-1)+R\geqslant N\}$,此 k 恰为构成R 叉 Huffman 树的内点个数。令 $t=k(R-1)+R-N$,表示构成满 R-叉树缺少的叶子数。在字符集中添加 t 个频数均为 0 的虚拟字符,这样对这 $N+t$ 个字符组成的字符集就可以构造一棵满R 叉树了。

2. 算法描述

将算法 9-5 很容易扩展为创建 R 叉 Huffman 树,描述成伪代码过程如下。

```
R-HUFFMAN(C,f,N,R)
k←min{k|k∈N 且 k(R-1)+R≥N}          ▷有 k+1 个内点
t←k(R-1)+R-N                       ▷t 个虚拟字符
Q←C                                ▷将 C 中 N 个字符插入优先队列 Q
insert t fictitious letter into Q
for i←1 to k+1                     ▷创建新的内点
    do allocate new node z
       f[z]←0
       for j←1 to R                ▷内点 z 的 R 个孩子
         do x←DEQUEUE(Q)
            z 的 children[j]←x      ▷children[j]为 z 的第 j 个孩子
            parent[x]←z
            f[z]←f[z]+f[x]
       ENQUEUE(Q,z)
return DEQUEUE(Q)
```

算法 9-6 算法 9-3 的扩展,创建 R 叉 Huffman 树

与算法 9-5 相比,算法 9-6 做了如下几点扩展。

(1) 第 2 行计算该 R 叉 Huffman 树的内点数 $k+1$。对于 $R=2$ 的特殊情形的算法 9-5 而言，这是不必的。因为 N 片叶子的满二叉树恰有 $N-1$ 个，即 k 是一个常数 $N-2$。

(2) 第 3 行和第 4 行将 $t(t=k(R-1)+R-N)$ 个虚拟字符插入到优先队列 Q 中。这在算法 9-5 中也是无须的。因为当 $R=2$ 时，总有 $k=N-1$ 使得 $k(R-1)+R=k+2=N-2+2=N$，这样 $t=0$。

(3) 第 6～12 行的 **for** 循环将从 Q 中弹出的 R 个结点依次作为新的内点 z 的孩子。而在算法 9-5 中 z 只有 $R=2$ 个孩子。

利用过程返回的 R 叉 Huffman 树 T，对树中每一片叶子沿着 parent 方向回溯到根，就可以计算出 C 中每个字符的编码，进而计算出平均存储量。描述成伪代码过程如下。

```
CALCULATE(T)
1  cost←sum←0
2  for each leaf x in T
3      do code[x]←∅
4         node←x, p←parent[node]
5         while p≠NIL
6             do j←node 作为 p 的孩子的编号
7                append j to code[x]
8                node←p, p←parent[p]
9         INVERSE(code[x])
10        cost←cost+f[x]·length[code[x]]
11        sum←sum+f[x]
12 return cost/sum
```

算法 9-7　根据 R 叉 Huffman 树计算字符编码及平均存储量的过程

算法中第 2～11 行的 **for** 循环对每一个叶子结点 x 进行处理。第 4 行设置 2 个变量 *node* 和 p 分别跟踪当前结点及其父结点。第 5～8 行的内嵌 **while** 循环自底向上逐层检测当前的 *node* 是其父结点的第几个孩子，记录为 j，追加到 x 的编码串 *code*。这样得到的 x 的编码是反向的，第 9 行调用 INVERSE 过程将 *code*[x]反转。第 10 行和第 11 行将 $f[x]\cdot length[code[x]]$ 累加到 *cost* 中，$f[x]$累加到 *sum* 中。算法最终在第 12 行将字符的平均存储量 *cost*/*sum* 返回。

9.2.3　程序实现

1. 数据类型定义

很容易想到用本书第 2 章中的程序 2-12 和程序 2-13 定义的 BTreeNode 来表示 Huffman 树。但那是表示二叉树的数据类型(它只有左右两个孩子)，用它来表示 R 叉 Huffman 树(它有 R 个孩子)显得有点力不从心。此外，Huffman 树一旦创建，无须做插入和删除操作。出于这些考虑，将 Huffman 树用数组表示成静态 R 叉树，树中每个结点存储为数组元素，内点的 R 个孩子用一个数组表示，存储孩子结点的下标，如图 9-9 所示。

图 9-9 所示为用数组静态表示满 R 叉树。图 9-9(a)是一棵具有 7 片叶子的满 3 叉树。

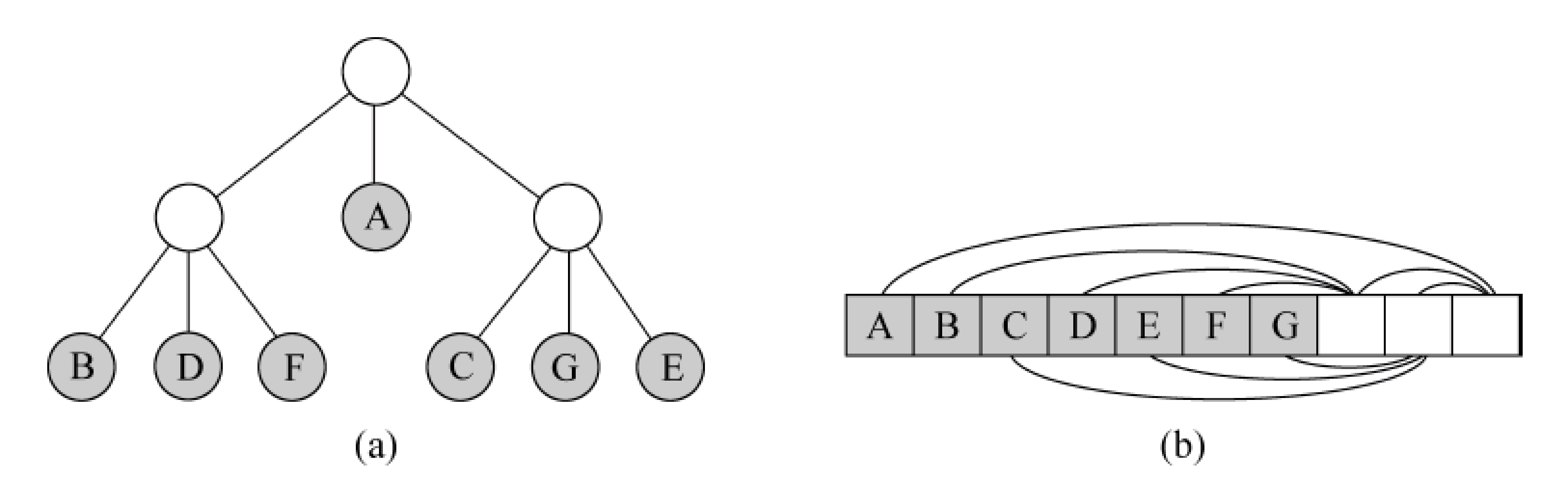


图 9-9　用数组静态表示满 *R* 叉树

带有阴影的表示叶子,白色结点是内点。图 9-9(b)是表示图 9-9(a)中 3 叉树的数组。数组中前 7 个元素表示树中的叶子结点,后面的元素表示内点,连接结点的弧线表示父子关系。其中,最后一个元素表示树根。

```
1 typedef struct{                                    /* Huffman 树结点类型 */
2     int ch;                                         /* 字符 */
3     int f;                                          /* 频数 */
4     int parent;                                     /* 父结点 */
5     int *child;                                     /* 孩子 */
6     char code[16];                                  /* 编码串 */
7 }Node;
8 typedef struct{                                     /* Huffman 树类型 */
9     Node *Nodes;                                    /* 结点数组 */
10     int N;                                         /* 叶子数 */
11     int R;                                         /* 基数 */
12 }HTree;
13 int Compare(Node** a,Node** b){
14     if((*a)->f==(*b)->f)                           /* 频数相等 */
15         return (int)((*b)->ch)-(int)((*a)->ch);    /* 比较字符 */
16     return (*b)->f-(*a)->f;                        /* 比较频数 */
17 }
```

程序 9-3　表示二叉树结点的数据类型 Node 和静态二叉树类型 HTree

对程序 9-3 的说明如下。

(1) 第 1～7 行定义了表示 Huffman 树中结点的数据类型 Node。作为结构体,它有 5 个数据属性。表示字符的 ch 属性,表示字符频数的 f 属性,表示父结点的 parent 属性,表示孩子数组的 child 属性以及表示字符编码串的 code 属性。

(2) 第 8～12 行定义了表示 Huffman 树的数据类型 HTree。它有 3 个数据属性。表示树中各个结点的数组属性 Nodes,表示树中叶子结点数的 N 属性以及表示基数(树中每个内点具有的孩子数)的 R 属性。

(3) 创建 Huffman 树时,由于要将结点加入到一个优先队列中,所以需要定义比较两个结点大小的函数。第 13～17 行定义的 Compare 就是这样的函数。该函数对两个由双重指针指引的结点类数据比较大小。第 14 行和第 15 行对检测到字符的频数相等情形按字符本身的编码值决定大小。第 16 行对不同的频数决定大小。

2. 创建 *R* 叉 Huffman 树

接下来实现算法 9-6，创建 *R* 叉 Huffman 树。

```
int calck(int r,int n){                 /* 对给定的 r 和 n 计算 min{k|k∈N 且 k(r−1)+r≥n} */
    int k=1;
    while(k*(r-1)+r<n)k++;
    return k;
}
HTree Huffman(char *C,int *f,int n,int r){
    HTree cTree;
    int i,j,k,t,z;
    PQueue *Q;
    Node *e;
    cTree.R=r; cTree.N=n;
    k=calck(r,n);                       /* k←min{k|k∈N 且 k(r−1)+r≥n},k+1 为内点数 */
    t=k*(r-1)+r;                        /* 叶子数 */
    Q=initPQueue(sizeof(Node *),t,Compare);         /* 创建优先队列 */
    cTree.Nodes=(Node *)malloc((t+k+1)*sizeof(Node));
    for(i=0;i<n;i++){                               /* 初始化 n 片字符叶子 */
        cTree.Nodes[i].ch=C[i];
        cTree.Nodes[i].f=f[i];
        cTree.Nodes[i].child=NULL;
        memset(cTree.Nodes[i].code,0,16);
    }
    for(i=n; i<t; i++){                             /* 初始化虚拟叶子结点 */
        cTree.Nodes[i].ch=n-t+1;
        cTree.Nodes[i].f=0;
        cTree.Nodes[i].child=NULL;
    }
    for(i=0;i<t;i++){                               /* 将所有叶子加入队列 */
        e=&(cTree.Nodes[i]);
        enQueue(Q,&e);
    }
    for(i=0,z=t;i<=k; i++,z++){                     /* 贪婪策略创建 Huffman 树 */
        cTree.Nodes[z].child=(int *)malloc(r*(sizeof(int)));
        cTree.Nodes[z].f=0;
        cTree.Nodes[z].ch='z'+z;
        for(j=0;j<r;j++){
            Node** x=(Node**)deQueue(Q);
            int self=*x-cTree.Nodes;
            cTree.Nodes[z].f+=(*x)->f;
            cTree.Nodes[z].child[j]=self;
            cTree.Nodes[self].parent=z;
        }
```

```
42          e=&(cTree.Nodes[z]);
43          enQueue(Q,&e);
44      }
45      pQueueClr(Q);
46      cTree.Nodes[--z].parent=-1;
47      return cTree;
48 }
```

程序 9-4　实现算法 9-6 的 C 函数

对程序 9-4 的说明如下。

(1) 第 6～48 行定义的函数 Huffman,实现算法 9-6 的过程 R-HUFFMAN。与算法过程一样,函数的 4 个参数 c、f、n、r 分别表示字符集数组、字符频数数组、字符个数和基数。函数返回表示该字符集 *R* 进制最优编码的 Huffman 树。

(2) 函数中第 7 行用程序 9-3 中定义的 *R* 叉 Huffman 树类型 HTree 定义了一个 Huffman 树对象 cTree,用作函数的返回值。第 9 行用本书第 3 章中程序 3-15～程序 3-18 实现的基于二叉堆的优先队列类型 PQueue 定义了一个优先队列对象 Q。第 12 行调用函数 calck(定义于第 1～5 行)针对给定的 n 和 r 计算用来表示内点数的 k。第 13 行计算表示叶子数的 t。第 15 行为 cTree 分配了存储结点的数组空间。

(3) 第 16～30 行实现算法 9-4 中的第 3 行和第 4 行的操作。其中,第 16～21 行初始化 n 片叶子结点。第 22～26 行初始化 t 片虚拟的字符叶子(有的话)。第 27～30 行将所有这些叶子结点加入到优先队列 Q 中。

第 31～44 行的 **for** 循环实现算法中第 5～13 行的 **for** 循环。其中,z 表示新的内点(下标),第 32～34 行实现算法中第 6 行的操作。注意,由于内点加入到 Q 中需要与其他结点比较大小,为使得该结点与叶子结点频数相等情况下,正常叶子应排列在前,故第 34 行将内点 z 的字符属性 ch 置为'z'+z,它比所有正常的字母字符都要大。当时当 z>5,则'z'+z 将超出 127,作为字符类型数据,它会表示成负数。所以,为了使结点大小比较能正确进行,程序 9-3 中将Node 中的表示结点字符属性的 ch 表示成 **int** 类型。

第 35～41 行的 **for** 循环对应算法过程中的第 8～12 行的 **for** 循环,完成设置 z 的 r 个孩子的操作。注意,第 37 行将变量 self 赋予从 Q 中弹出的结点 x(它是 cTree 的 Nodes 数组中的一个元素的地址)与 cTree 的 Nodes 属性的差。这实际上计算出 x 在 Nodes 数组中的下标。第 39 行将此 self 记录 x 作为 z 的第 j 个孩子,第 40 行记录 x 的父结点为 z。

第 46 行将根结点的 parent 属性置为－1,与其他结点加以区别。

3. 计算字符编码及平均存储量

利用函数 Huffman 创建的 *R* 叉 Huffman 树,可以实现计算字符 *R* 进制编码及平均存储量的算法过程。

```
void inverse (char *s){
    int i=0,j=strlen(s)-1;
    while(i<j){
        swap(s+i,s+j,sizeof(char));
```

```
5           i++;j--;
6       }
7 }
8 double Calculate(HTree T){
9     int i,sum=0;
10    double cost=0.0;
11    for(i=0;i<T.N;i++){
12        int j,node=i,p=T.Nodes[i].p,index=0;
13        while(p!=-1){
14            j=0;
15            while(T.Nodes[p].child[j]!=node)j++;
16            T.Nodes[i].code[index++]='0'+j;
17            node=p;
18            p=T.Nodes[node].parent;
19        }
20        inverse (T.Nodes[i].code);
21        cost+=T.Nodes[i].f* strlen(T.Nodes[i].code);
22        sum+=T.Nodes[i].f;
23    }
24    return cost/sum;
25 }
```

程序 9-5　实现算法 9-7 的 C 函数

对程序 9-5 的说明如下。

(1) 第 8～25 行定义的函数 Calculate 实现算法 9-7 中的 CALCULATE 过程。与算法一样，函数具有一个表示 Huffman 树的参数 T，并返回字符的平均存储量(**double** 型数据)。

(2) 函数中第 11～23 行的 **for** 循环对应算法过程中的第 2～11 行的 **for** 循环，对第 *i* 个叶子结点计算其编码串。其中，变量 node、p 的意义与算法中的一致，只不过此处表示当前结点的下标和当前结点父结点下标。为跟踪第 i 个叶子结点的编码串尾，第 12 行设置变量 index，初始化为 0。第 13～19 行的 **while** 循环对应算法过程中的第 5～8 行的 **while** 循环，从自底向上地沿 parent 域计算第 i 个叶子结点的编码串。其中，第 15 行的 **while** 循环在 p 的孩子数组 child 中，寻找 node 的下标 j，实现算法中的第 7 行操作。第 16 行将 j 追加到第 i 个叶子结点的编码串尾 code[index]。由于在程序 9-4 的第 20 行将 code 数组中的每个元素初始化为 0(也就是字符'\0')，故在第 13～19 行的 **while** 循环完成后无须再对 code 的末尾添加'\0'了。

(3) 第 20 行调用函数 inverse 将第 i 个结点的编码串反转。该函数定义于第 1～7 行。

4. 主函数

下列是调用程序 9-4 和程序 9-5 定义的函数解决 Variable Radix Huffman Encoding 问题的 main 函数。

```
int main(){
```

```
2    int r,n,count=1;
3    FILE *f1=fopen("chap09/Variable Radix Huffman Encoding/inputdata.txt","r"),
4         *f2=fopen("chap09/Variable Radix Huffman Encoding/outputdata.txt","w");
5    assert(f1&&f2);
6    fscanf(f1,"%d%d",&r,&n);
7    while(r){
8         int i,*f;
9         char *C;
10        HTree T;
11        double ave;
12        assert(C=(char*)malloc(n*sizeof(char)));
13        assert(f=(int*)malloc(n*sizeof(int)));
14        for(i=0;i<n;i++){
15            C[i]='A'+i;
16            fscanf(f1,"%d",f+i);
17        }
18        T=Huffman(C,f,n,r);
19        ave=Calculate(T);
20        fprintf(f2,"Set %d; average length %.2f\n",count,ave);
21        for(i=0;i<T.N;i++)
22            fprintf(f2,"%c: %s\n",T.Nodes[i].ch,T.Nodes[i].code);
23        fprintf(f2,"\n");
24        free(C);free(f);
25        clrHTree(&T);
26        fscanf(f1,"%d%d",&r,&n);
27    }
28    fclose(f1);fclose(f2);
29    return 0;
30 }
```

程序 9-6　解决 Variable Radix Huffman Encoding 的 main 函数

对程序 9-6 的说明如下。

(1) 第 3～5 行打开输入、输出文件 f1 和 f2。第 6 行在 f1 中读取第一个案例的基数 r 和字符数 n。第 7～27 行的 while 循环处理 f1 中的每一个案例。

(2) 第 12 行和第 13 行为本案例的字符数组 C 和频数数组 f 分配空间，第 14～17 行初始化第 i 个字符 C[i]并从 f1 中读取该字符的频数 f[i]。第 18 行调用函数 Huffman 创建本案例的 Huffman 树 T，第 19 行调用函数 Calculate 计算每个字符的编码串及字符的平均存储量 ave。第 20～23 行向文件 f2 写入计算的结果。第 26 行从 f1 中读取下一个案例的基数 r 和字符数 n。

程序 9-3 和程序 9-4 存储在 greedy 文件夹中的 huffman.h 和 huffman.c 中。程序 9-5 和程序 9-6存储为 chap09/Variable Radix Huffman Encoding 文件夹中的文件 VariableRadixHuffmanEncoding.c 中。

9.3　最小生成树

9.3.1　算法描述与分析

1. 问题的理解与描述

设 $G=\langle V,E\rangle$ 是一个无向连通图，n 个顶点用前 n 个正整数编号，即 $V=\{1,2,\cdots,n\}$。G 具有权函数 $w: E\to\mathbf{R}$[①]。目标是找到 G 的一棵生成树[②] T，使得其权 $w(T)$——树中每条边的权之和——最小。本问题可以形式化为如下。

输入：图 $G=\langle V,E\rangle$ 及权函数 $w: E\to\mathbf{R}$。

输出：G 的一棵生成树 T，使得 $w(T)$ 最小。

2. 最优子结构

这是一个组合优化问题，G 有若干棵生成树，每棵生成树 T 均有其权值 $w(T)$ 作为目标值。目的是求得具有最小目标值的生成树。这个问题的最优子结构阐述如下。

设 T 是 G 的一棵最小生成树，e 是 T 中一条边，删去 e 则将 T 分成两部分 T_1 和 T_2。假定 V_1 和 V_2 分别是 T_1 和 T_2 所包含的顶点集，G_1 和 G_2 分别是 G 的 V_1 和 V_2 诱导的子图[③]，则 T_1 和 T_2 分别是 G_1 和 G_2 的最小生成树。

3. 贪婪选择性

本问题的贪婪选择性质可以阐述如下。

定理 9-3　设 $G=\langle V,E\rangle$ 是一个无向图且具有权函数 $w: E\to\mathbf{R}$，U 是 V 的一个真子集。则

(1) 在集合 $\{(x,y)\mid(x,y)\in E, x\in U, y\in V-U\}$ 中找出权值最小者，记为 (u,v)。则 (u,v) 必在 G 的一棵最小生成树中。

(2) 若记 G' 是 G 中由 U 诱导的子图，T 是 G' 的最小生成树，则把(1)中选择的边 (u,v) 添加到 T 中将构成 G 的由 $U\cup\{v\}$ 诱导的子图的最小生成树。

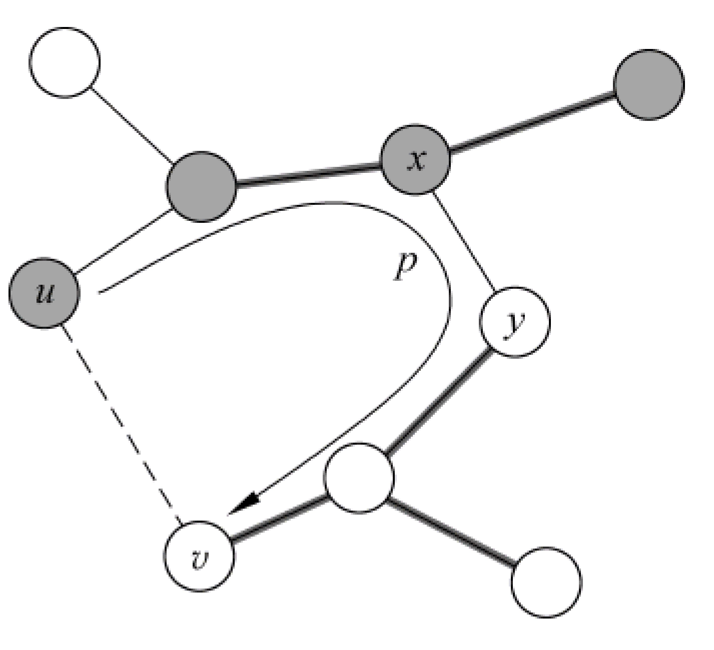


图 9-10　定理 9-3 的证明思路

本定理(1)的正确性可说明如下。若否，设 (u,v) 不在 G 的任一最小生成树中。设 T 为 G 之一最小生成树，将 (u,v) 添加到 T 中必在其中形成一条包含 (u,v) 的圈 p（见图 9-10）。在此 p 中，至少存在一条边 (x,y)，使得

---

① $\mathbf{R}$ 表示实数集合。

② 图 G 的生成树指的是 G 的一个连接 G 的所有顶点的连通子图，其中没有圈。

③ 若 $V'\subseteq V$ 且 $E'\subseteq E$，图 $G'=(V',E')$ 是 $G=(V,E)$ 的一个**子图**。给定一个集合 $V'\subseteq V$，G 的一个由 V' **诱导**的子图是子图 $G'=(V',E')$，其中 $E'=\{(u,v)\in E: u,v\in V'\}$。

$x \in U, y \in V-U$(否则不会形成一个圈)。在此圈中删除(x,y)将得到一棵生成树 T',T'与 T 相比仅仅只有一条边(x,y)与(u,v)不同。于是 $w(T')=w(T)-w(x,y)+w(u,v)<w(T)$,此与$T$ 为G 之一最小生成树矛盾。

对本定理的(2),首先说明 $T \cup \{(u,v)\}$仍然构成一棵根树。这是因为 $u \in U$,而 $v \notin U$,因此U 中各顶点通过u 可达v,且(u,v)不会与 T 中的边构成圈。所以,$T \cup \{(u,v)\}$是一棵树。其次,根据(1)中(u,v)的选取可知 $T \cup \{(u,v)\}$是 G 的由 $U \cup \{v\}$诱导的子图中权值最小的生成树。

4. 算法的伪代码描述

根据最优子结构特性和贪婪选择性质,可以用如下的贪婪选择策略,来解决构造权函数为 w 的图G 的以r 为根的最小生成树问题。

从 $U=\{r\}$开始,在$\{(x,y) \mid (x,y) \in E, x \in U, y \in V-U\}$中找权值最小的边$(u,v)$,将$(u,v)$加入到 T 中,v 加入到U 中。这个方法称为 Prim 算法。算法的运行如图 9-11 所示。

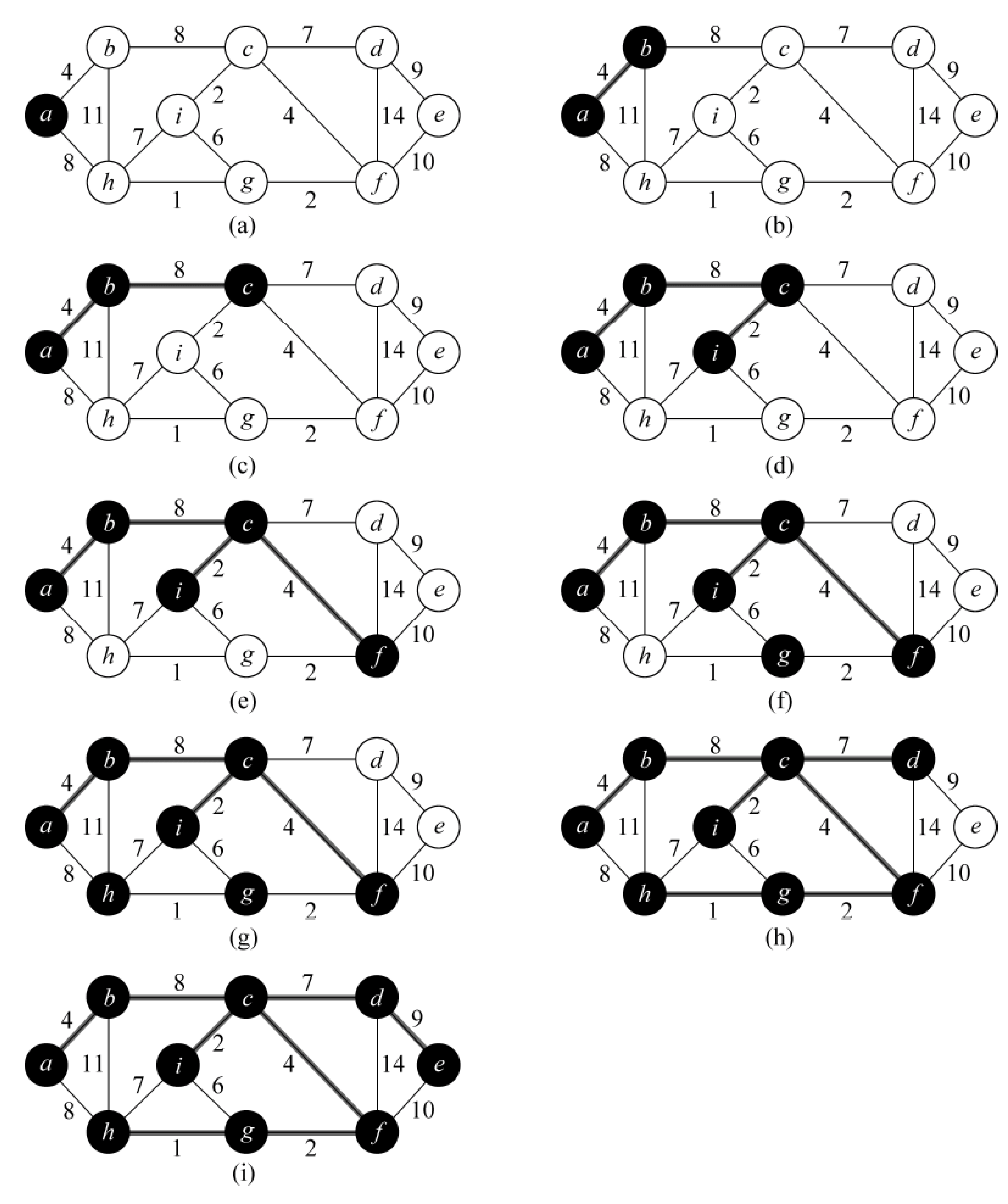


图 9-11 构造带权图的最小生成树的 Prim 方法

```
MST-PRIM(G,w,r)
1 U←{r}
2 T←∅
3 Q←V－{r}
4 while Q≠∅
5     do (u,v)←集合{(x,y)|(x,y)∈E,x∈U,y∈Q}中权值最小的边
6        T←T∪{(u,v)}
7        U←U∪{v}
8        Q←Q－{v}
9 return T
```

算法 9-8 解决最小生成树问题的 Prim 算法

5. 算法的正确性

算法中第 4～8 行的 **while** 循环具有如下的循环不变量：每次重复之初，边集 T 构成 G 的由 U 诱导的子图的最小生成树，且 Q 中保存的是顶点集 $V-U$。

利用此循环不变量，可以证明算法的正确性。

对 U 中元素个数 $|U|$（该循环每重复一次，U 中元素恰增加一个）进行数学归纳。初始时，由第 1 行可见 U 为单元素集$\{r\}$，$|U|=1$。由于 G 中由 U 诱导的子图是平凡的，其生成树必为空。所以第 2 行将 T 初始化为空集符合上述不变量。注意第 3 行 Q 初始化为 $V-U$。

假定 $|U|=i$ 时本次循环重复之初，不变量为真，即边集 T 构成 G 的由 U 诱导的子图的最小生成树，且 $Q=V-U$。在本次重复中，第 5 行选取$\{(x, y)|(x, y)(E,x(U,y(Q\}$中权值最小的边为$(u, v)$。第 6 行将$(u, v)$添加到 T 中，第 7 行将 v 添加到 U 中，第 8 行通过从其中删除 v 将 Q 调整为 $V-U$。根据定理 9-3 将(u, v)添加到 T 中将构成 G 的由 $U(\{v\}$诱导的子图的最小生成树。此状态将维持到下一次重复之初。即$|U|=i+1$ 时的重复之初，不变量依然为真。

循环终止时，Q 为空集，U 为 V，根据循环不变量知 T 是 G 中由 $U(=V)$诱导的子图——此时就是 G 的最小生成树。

上述过程的实现难点在于第 5 行从集合$\{(x,y)|(x,y)\in E,x\in U,y\in Q\}$中选取权值最小的边。为每个顶点 u 增添两个属性：属性 $key[u]$记录该顶点处于 $Q(Q=V-U)$中时与 U 中顶点构成的边中的最小权值，属性 π 记录该顶点处于 Q 中时与 U 中构成最小权值边的另一个顶点，也就是在最小生成树中的父结点。同时将 Q 改造成一个最小优先队列，Q 中顶点以 key 作为其优先级。初始时，所有顶点的 π 属性指向空，Q 置为 V，$key[r]$为 0，其余顶点的 key 属性皆置为∞。每次从 Q 中出队的顶点 u 为 key 属性最小的顶点（第一个出队的为 r），以此保证所选的是 U 与 Q 之间权值最小的边。扫描 Q 中与 u 相邻的所有顶点，调整这些顶点 key 和 π 属性。只要重复 n 次上述操作就构造了 G 的一棵以 r 为根的最小生成树。由于 π 属性跟踪了生成树中结点的父子关系，所以省略了集合 T。key 属性跟踪了 Q 中顶点目前与 U 中顶点构成边的最小权值，所以 U 也可省略。细化后的代码如下。

```
MST-PRIM(G,w,r)
1 for each u∈V[G]
2     do key[u]←∞
```

```
3             π[u]←NIL
4 key[r]←0
5 Q←V[G]
6 while Q≠∅
7       do u←DEQUEUE(Q)
8          for 每个与 u 相邻的顶点 v
9             do if v∈Q 且 w(u,v)<key[v]
10                  then π[v]←u
11                       key[v]←w(u,v)
12          FIX(Q)
13 return key and π
```

算法 9-9　Prim 算法的细化

本算法中的优先队列 Q 的使用与 HUFFMAN 算法中的不同。在那里,元素从进入队列到离开队列,其键值不会发生变化。而在此处,第 11 行的操作是对留驻在队列 Q 中的元素的键值进行改写。这样,每当第 8～11 行的 **for** 循环重复一次,就可能破坏队列 Q 的堆性质,所以在第 12 行要对 Q 做一次堆性质的维护操作 FIX(Q)。

一般说来,一个图的最小生成树不必是唯一的。当队列中键值最小的元素不止一个时,不同的队首将使第 7 行的出队操作得到不同的顶点,这可能导致不同的生成树。但是所有的最小生成树都具有相同的权值。

跟算法 9-8 相比,由于省略了边集 T(最终形成 G 的最小生成树),所以第 13 行返回的不再是 T 而是能反映最小生成树 T 的所有信息的顶点属性数组 *key* 和 π。

6. 算法的运行时间

粗略地看,算法 MST-PRIM 的第 1～3 行耗时 $\Theta(n)$,第 5 行创建优先队列耗时 $\Theta(n)$,第 6～12 行的 **while** 循环重复 n 次,每次重复中第 8～11 行的 **for** 循环耗时 $\Theta(n)$。第 12 行对优先队列的维护(相当于重建二叉堆)耗时 $\Theta(n)$,所以该算法的总耗时为 $\Theta(n^2)$。

9.3.2　程序实现

1. 数据准备

如上所述,应当改造优先队列 PQueue,使其能够对经过改写了队列中元素键值后的队列维护堆性质。实际上只要在第 3 章创建的程序文件 pqueue.c 加入以下的内容,并将所定义的函数 fix 的原型声明加入到 pqueue.h 中就可以了。

```
void fix(PQueue *q){
    buildHeap(q->heap,q->eleSize,q->heapSize,q->compare);
}
```

函数 fix 调用 buildHeap,按照队列 q 的元素大小比较规则 compare 重新将 q 创建成一个堆。

由于插入队列中的元素是指向 key 数组元素的指针,所以不能直接引用先前定义过关于 double 型数据的比较规则,需要定义一个能通过双重指针比较 **double** 型数据的函数:

```
int dblLess(double **x,double **y){
    if((**x)<(**y))
        return 1;
    if((**x)>(**y))
        return -1;
    return 0;
}
```

程序 9-7　比较优先队列中 double 型指针元素指向的数据大小的函数

把函数定义存储在 Utility 目录的头文件 compare.c 中，以备重用。

此外，还需要考虑如何表示一个图。与第 8 章中 8.4 节表示带权有向图相似，对于一个带权无向图 $G=\langle V,E\rangle$，权函数 $w: E\rightarrow\mathbf{R}$，通常用 G 的权矩阵 $W(w_{ij})_{n\times n}$来表示。其中

$$w_{ij}=\begin{cases}w(i,j) & (i,j)\in E\\ 0 & (i,j)\notin E\end{cases}$$

同样，为便于作为参数传递，用一维数组按行优先规则来表示一个二维数组，用以表示一个带权图。为方便计，在程序中把图中的 n 个顶点表示为 $V=\{0,1,\cdots,n-1\}$，这样就和 key 等数组的下标统一起来了。

2. 实现函数

利用上述的数据准备，将算法 9-9 实现为如下 C 函数。

```
pair mstPrim(double *w,int n,int r){
    int *pi=(int*)malloc(n*sizeof(int)),
        *poped=(int*)malloc(n*sizeof(int)),
         i,u,v;
    double *key=(double*)malloc(n*sizeof(double));
    PQueue *Q=initPQueue(sizeof(double*),n,dblLess);
    for(i=0;i<n;i++){
        key[i]=DBL_MAX;
        pi[i]=-1;
        poped[i]=0;
    }
    key[r]=0.0;
    for(i=0;i<n;i++){
        double *e;
        e=&key[i];
        enQueue(Q,&e);
    }
    while(!empty(Q)){
        u=(*(double**)deQueue(Q))-key;
        poped[u]=1;
        for(v=0;v<n;v++){
            if(w[u*n+v]>0.0)
                if(!poped[v])
```

```
24                  if(w[u * n+v]<key[v]){
25                      pi[v]=u;
26                      key[v]=w[u * n+v];
27                  }
28          }
29          fix(Q);
30      }
31      pQueueClr(Q);
32      return make_pair(key,pi);
33 }
```

程序 9-8　实现算法 9-9 的 C 源代码

对程序 9-8 的说明如下。

(1) 函数 mstPrim 除了表示图 *G* 的数组 w 和作为根的顶点编号 r 两个参数外,还有一个表示图的顶点数 n 的参数。返回值类型是 8.3.3 节中开发的数据类型 pair,其中封装了计算所得的数组 key 和 pi。

(2) 第 2 行和第 3 行声明了两个整型数组:pi(对应于过程 MST-PRIM 中的 π)和用来跟踪顶点是否在队列中 poped。第 5 行声明了跟踪各顶点的 key 属性的数组 key。第 7~11 行的 **for** 循环完成对这 3 个数组的初始化工作。第 12 行将根结点 r 的 key 值置为 0,作为优先队列的首元素。所有这些代码实现 MST-PRIM 过程中的第 1~4 行的操作。第 6 行声明了 PQueue 类型的指针对象 Q,并对其进行了初始化,将程序 9-8 定义的函数 dblLess 作为优先队列中元素大小比较规则。

(3) 第 13~17 行代码实现 MST-PRIM 过程中第 5 行的建立优先队列操作 $Q \leftarrow V[G]$。要注意的是,由于加入到队列 Q 中的是指向数组 key 的元素的指针,若省掉第 14 行和第 15 行,直接调用 enQueue(Q,&key[i]),则按照第 3 章程序 3-17 的解析,执行的结果是将 &key[i] 指向的 key[i]加入到 Q 中,而不是把 &key[i]加入进去。所以需要有一个存储 &key[i]的变量e 来做中转,这就是第 14~15 行的工作。

(4) 第 18~30 的代码实现 MST-PRIM 过程中第 6~12 行的操作。其中,第 19 行调用函数 deQueue 将 Q 的队首元素出队,解析出其中存储的指向数组 key 的元素的地址,与 key 的首地址之差,恰好是该元素在 key 中的下标值,赋值给表示图中对应顶点的 u。第 20 行将出队的顶点 u 做出队标志(poped[u]置为 1),这个标志是第 23 行所要检测的。

此外,请注意第 22 行、第 24 行和第 26 行对由参数传递进来的数组 w 的元素 w[u * n+v] 的访问相当于若 w 是一个二维数组对元素 w[u][v]的访问。

(5) 第 38 行将计算所得的数组 key 和 pi 封装到 pair 对象中,并将其返回。

为便于代码重用,将程序 9-8 中的函数定义存储在 greedy 文件夹中的源文件 prim.c 中,而将该函数的原型声明存储于头文件 prim.h 中。

9.3.3　应用——北方通信网

Arctic Network

The Department of National Defence (DND) wishes to connect several northern

outposts by a wireless network. Two different communication technologies are to be used in establishing the network: every outpost will have a radio transceiver and some outposts will in addition have a satellite channel.

Any two outposts with a satellite channel can communicate via the satellite, regardless of their location. Otherwise, two outposts can communicate by radio only if the distance between them does not exceed D, which depends of the power of the transceivers. Higher power yields higher D but costs more. Due to purchasing and maintenance considerations, the transceivers at the outposts must be identical; that is, the value of D is the same for every pair of outposts.

Your job is to determine the minimum D required for the transceivers. There must be at least one communication path (direct or indirect) between every pair of outposts.

The first line of input contains N, the number of test cases. The first line of each test case contains $1 \leqslant S \leqslant 100$, the number of satellite channels, and $S < P \leqslant 500$, the number of outposts. P lines follow, giving the (x, y) coordinates of each outpost in km (coordinates are integers between 0 and 10,000). For each case, output should consist of a single line giving the minimum D required to connect the network. Output should be specified to 2 decimal points.

Sample Input

```
1
2 4
0 100
0 300
0 600
150 750
```

Sample Output

```
212.13
```

1. 问题描述与分析

要将北方各哨所用两种无线技术实现通信连接。P 个哨所(每个哨所都用坐标标识)中有 S 个可以使用卫星通信,卫星通信不受地理位置的限制。P 个哨所中的每一个都有无线收发机用来与其他哨所联系,但无线电通信受接收器功率的影响:功率越大,通信覆盖半径 D 越大,当然代价也越大。为方便维护与管理,要用统一型号的无线收发机配置各哨所。要使任意两个哨所之间能进行直接或间接的通信,并使得装备费用最低。问题可形式化为如下。

输入: P 个哨所的坐标:$A=\{(x_1, y_1), \cdots, (x_P, y_P)\}$。可以使用卫星通信的哨所数 S。

输出: 连接所有哨所的通信网络中所配备无线收发器的最小覆盖半径 D。

将每个哨所视为图中一个顶点,哨所间的距离作为两顶点间边的权值。这样将得到一个具有 P 个顶点的带权完全图。作为连接所有哨所的最节省通信网应当是该完全图的最

小生成树。但还需要考虑在最小生成树中有 S 个顶点可以用卫星频道相互通信,这样,就可以在最小生成树中对最长的若干条边的端点配置卫星通信,剩下的边中最长者就是这棵生成树对应的配备方案的无线收发机的最小代价 D 了。

2. 算法描述

首先需要将案例中 P 个哨所坐标转换成一个带权无向图。假定哨所坐标表示成了两个数组 $x[1..P]$、$y[1..P]$,可用下列过程转换成一个无向带全图的权矩阵 $w[1..P,1..P]$。

MAKE-GRAPH(x,y,P)

for $i \leftarrow 1$ **to** P

  do for $j \leftarrow 1$ **to** i

    do $w[i,j] \leftarrow w[j,i] \leftarrow \sqrt{(x[i]-x[j])^2+(y[i]-y[j])^2}$

return w

算法 9-10 将 P 个哨所的坐标转换成一个无向带全图

对表示图 G 的权矩阵 w 计算最小的无线传输距离 D 的过程表示为如下的伪代码。

ARTIC-NETWORK(w,S)

$T \leftarrow$ MST-PRIM(w,1) ▷ T 为以 1 为根的最小生成树

$n \leftarrow S$

$Q \leftarrow T$ 中所有的边 ▷ Q 为一最大优先队列

while $n \geqslant 1$ ▷ 配置卫星频道

  do $(u,v) \leftarrow$ DEQUEUE(Q)

   if u 未访问过

    then $n \leftarrow n-1$

   if v 未访问过

    then $n \leftarrow n-1$

if $n=0$

  then $(u,v) \leftarrow$ DEQUEUE(Q)

return $w[u,v]$ ▷ 无线收发机覆盖半径

算法 9-11 计算无线电最小传输距离的过程

3. 程序实现

```
#include "../../datastructure/pqueue.h"
#include "../../greedy/prim.h"
double *w;                                          /* 图的权矩阵 */
int *pi;                                            /* 最小生成树 */
int p;                                              /* 哨所数 */
int s;                                              /* 安装卫星通信机的哨所数 */
int compare(const int **a,const int **b){           /* 最小生成树中边的大小比较 */
    int u=**a,v=*a-pi,                              /* (u,v)为由 a 确定的边 */
        x=**b,y=*b-pi;                              /* (x,y)为由 b 确定的边 */
    if(w[u*p+v]>w[x*p+y])                           /* (u,v)比(x,y)长 */
        return 1;
```

```
    if(w[u*p+v]<w[x*p+y])                    /*(u,v)比(x,y)短*/
        return -1;
    return 0;                                /*(u,v)和(x,y)等长*/
}
double *makeGraph(int *x,int *y){
    int i,j;
    double *w=(double*)malloc(p*p*sizeof(double));
    for(i=0;i<p;i++)
        for(j=0;j<=i;j++)
            w[i*p+j]=w[j*p+i]=sqrt((double)(x[i]-x[j])*(x[i]-x[j])+
                                      (y[i]-y[j])*(y[i]-y[j]));
    return w;
}
double ArcticNetwork(double *w,int p,int s){
    PQueue *Q=initPQueue(sizeof(int**),p-1,compare);
    int *e,i,n=s, *visited=(int*)calloc(p,sizeof(int)),u,v;
    double d;
    pair t=mstPrim(w,p,0);
    pi=(int*)(t.second);
    for(i=0;i<p;i++){                        /*向优先队列 Q 加入最小生成树中所有的边*/
        if(pi[i]!=-1){
            e=pi+i;
            enQueue(Q,&e);
        }
    }
    while(n>=1){
        v=*((int**)deQueue(Q))-pi;
        u=pi[v];
        if(!visited[u]){
            visited[u]=1;
            n--;
        }
        if(!visited[v]){
            visited[v]=1;
            n--;
        }
    }
    if(!n){
        v=*((int**)deQueue(Q))-pi;
        u=pi[v];
    }
    d=w[u*p+v];
    pQueueClr(Q); free(w);free(pi);free(visited);
    return d;
}
```

```
56 int main(){
57     int i,n;
58     FILE *f1,*f2;
59     assert(f1=fopen("chap09/Arctic Network/inputdata.txt","r"));
60     assert(f2=fopen("chap09/Arctic Network/outputdata.txt","w"));
61     fscanf(f1,"%d",&n);
62     for(i=0;i<n;i++){
63         int j,*x,*y;
64         fscanf(f1,"%d%d",&s,&p);
65         assert(x=(int*)malloc(p*sizeof(int)));
66         assert(y=(int*)malloc(p*sizeof(int)));
67         for(j=0;j<p;j++)
68             fscanf(f1,"%d%d",x+j,y+j);
69         w=makeGraph(x,y);
70         fprintf(f2,"%.2f\n",ArcticNetwork(w,p,s));
71         free(x);free(y);
72     }
73     fclose(f1);fclose(f2);
74     return 0;
75 }
```

程序 9-9　解决 Arctic Network 问题的 C 程序

对程序 9-9 的说明如下。

(1) 第 3～6 行定义的变量 w、pi、p、s 分别表示一个案例中指向无向带权图权矩阵的指针,指向调用 mstPrim 以后返回的最小生成树中各结点的数组指针,哨所数和安装卫星通信器的哨所个数。之所以将这些数据定义成全局变量,是因为比较生成树中边的大小关系时需要用到这些数据,而比较函数的原型又是严格定义好的仅具有 2 个 **void** 型指针的函数。见第 7～15 行定义的函数 compare。

(2) 第 16～23 行定义的函数 makeGraph 实现算法 9-10,将案例数据转换成无向带权图的权矩阵。代码结构与算法过程几乎一致,读者可对比阅读。第 24 行和第 25 行定义的函数 ArcticNetwork 实现算法 9-11,完成一个案例的计算。该函数维护一个能存储 p−1 个元素的优先队列,用来存放图的最小生成树的所有边。第 30～35 行将第 28 行和第 29 行调用 mstPrim 计算出来的最小生成树数组 pi 中除了根以外的所有元素一一加入优先队列 Q 中,完成算法中第 3 行的操作。注意,由于 pi 数组中的元素 pi[i]表示树中边(pi[i],i)。其长度为 w[p[i],i],所以 Q 中比较元素优先级的规则由第 7～15 行定义的函数 compare 确定。函数中还维护了一个用来表示顶点是否已经访问过的数组 visited,visite[i]的值为 0,表示顶点 i 未曾访问过,值为 1 表示 i 已经被访问过。利用表示安装卫星通信器的哨所数的变量 n(初始化为 s),第 36～47 的 **while** 循环对应算法中第 4～9 行的 **while** 循环反复从 Q 中弹出若干条边(直至关联这些边的顶点个数小于 1 为止)。第 54 行将最后从 Q 中弹出的边的长度作为计算结果返回。

(3) 第 56～75 行的 main 函数在第 58～60 行打开输入、输出文件 f1 和 f2。第 61 行从 f1 中读取案例数 n,第 62～72 行的 **for** 循环处理每一个案例。其中,第 64 行 f1 中读取本案

例的哨所数 p 和安装卫星通信器的哨所数 s。第 67 行和第 68 行读取 f1 中的各哨所坐标数据，第 69 行调用函数 makeGraph 将其转换成图的权矩阵 w，第 70 行将调用 ArcticNetwork 的返回值写入 f2 中。

程序 9-9 存储于 chap09/Arctic Network/中的源文件 ArcticNetwork.c。

9.4　单源最短路径问题

9.4.1　算法描述与分析

1. 问题的理解与描述

有向带权图 $G=\langle V,E\rangle$，$V=\{1,2,\cdots,n\}$，$E\subseteq V\times V$。其权函数 w：$E\rightarrow\mathbf{R}^+$ 将边映射到一个非负实数权值。路径 $p=\langle v_0,v_1,\cdots,v_k\rangle$的**权**是构成它的各条边的权之和：

$$w(p)=\sum_{i=1}^{k}w(v_{i-1},v_i)$$

按

$$\delta(u,v)=\begin{cases}\min\{w(p):u\overset{p}{\leadsto}v\} & \text{若有 } u \text{ 到 } v \text{ 的路径}\\ \infty & \text{否则}\end{cases}$$

定义从 u 到 v 的**最短路径权**。于是从顶点 u 到 v 的**最短路径**定义为路径 p，其权值 $w(p)=\delta(u,v)$。单源最短问题指的是对于图中一个顶点(源)$s\in V$，计算出从 s 出发到其余每一个顶点 $v\in V-\{s\}$的最短路径及其权值 $\delta(s,v)$。形式化为如下。

输入：有向带权图 $G=\langle V,E\rangle$，其权函数 w：$E\rightarrow\mathbf{R}^+$，源顶点 $s\in V$。

输出：从 s 到各顶点 v_i 的最短路径及数组$\{\delta(s,v_1),\delta(s,v_2),\cdots,\delta(s,v_n)\}$。

2. 最优子结构

对于每一个顶点 $v\in V-\{s\}$而言，这是一个组合优化问题。s 到 v 可能有若干条路径，每条路径都以各自的权值作为目标值。目的是计算出目标值最小的路径。单源最短路径问题的最优子结构性质阐述如下。

给定一个带权有向图 $G=(V,E)$，其权函数 w：$E\rightarrow\mathbf{R}^+$，设 $p=\langle v_1,v_2,\cdots,v_k\rangle$为从顶点 v_1 到顶点 v_k 的一条最短路径，对任意满足 $1\leqslant i\leqslant j\leqslant k$ 的 i 和 j，设 $p_{ij}=\langle v_i,v_{i+1},\cdots,v_j\rangle$为 p 中从顶点 v_i 到顶点 v_j 的子路径。则 p_{ij}是从 v_i 到 v_j 的一条最短路径。

3. 贪婪选择性

本问题的贪婪选择性质如定理 9-4 所述。

定理 9-4　设集合 $S\subseteq V$，$S\neq V$(初始时 $S=\varnothing$)。对每个 $v\in V-S$，维护属性 $d[v]$：从源点 s 出发，其间仅经过 S 中顶点到达 v 的最短路径的权(初始时 $d[s]=0$，对 $v\neq s$，$d[v]=\infty$)。若从 $V-S$ 中选取 $d[u]$最小的项点 u 加入到 S 中，则 $d[u]=\delta(s,u)$。

事实上，可以对 S 中的顶点个数做归纳。当$|S|=0$ 时，$S=\varnothing$，$V-S$ 中从 s 出发仅经过

S 中顶点到达的顶点 d 值最小的就是 s($d[s]=0$,其他的顶点 v 均有 $d[v]=\infty$)最短所以将有 s 进入 S。由于 $\delta(s,s)=0$,所以,此时有 $d[s]=\delta(s,s)$。假定 $|S|=k-1$ 时,$V-S$ 中从 s 出发仅经过 S 中顶点到达的顶点 d 值最小的是 u,且 $d[u]=\delta(s,u)$。将此 u 从 $V-S$ 移入 S 中,此时,S 中的顶点都已确定了从 s 出发的最短路径,最短路径距离就是各自的 d 值。对 $V-S$ 中只有那些与 u 相邻的顶点 v 其 d 值可能发生变化。按如下的方法修改这些顶点的 d 值:

$$d[v]=\begin{cases}d[u]+w(u,v) & \text{若 } d[v]>d[u]+w(u,v)\\ d[v] & \text{否则}\end{cases}$$

显然此时 $V-S$ 中的顶点 v 从 s 出发仅经过 S 中顶点到达的路径距离为 $d[v]$,$|S|=k$,若在 $V-S$ 中选取 d 值最小的顶点 u,下面证明 $d[u]=\delta(s,u)$。首先,显然有 $\delta(s,u)\leqslant d[u]$。设 s 到 u 的一条最短路径为 p,从 u 起反向在此路径行进,进入 S 前的最后一个顶点设为 y,y 的下一个顶点设为 x。则 p 分成三段:s　$x\to y$　u,如图 9-12 所示。

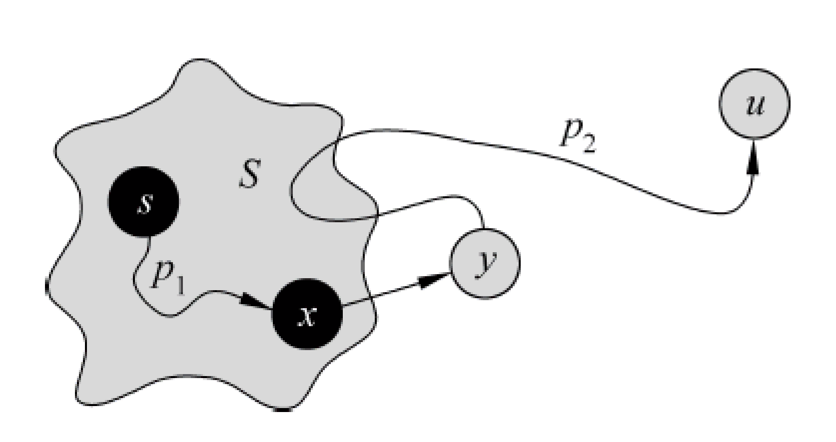


图 9-12　s 到 u 的最短路径

显然 p 的权值为

$$\begin{aligned}\delta(s,u)&=\delta(s,y)+p_2\text{ 的权值} && \text{(最优子结构)}\\ &\geqslant \delta(s,y) && (p_2\text{ 的权值}\geqslant 0)\\ &=\delta(s,x)+w(x,y) && \text{(最优子结构)}\\ &=d[x]+w(x,y) && (x\text{ 在 }S\text{ 中})\\ &\geqslant d[y] && \text{(根据 }x\text{ 进入 }S\text{ 时对 }d[y]\text{ 的调整)}\\ &\geqslant d[u] && \text{(根据 }u\text{ 的选择)}\end{aligned}$$

于是我们得到 $\delta(s,u)\geqslant d[u]$,连同 $\delta(s,u)\leqslant d[u]$,得到 $d[u]=\delta(s,u)$。

4. 算法的伪代码描述

利用这个贪婪性质,可以得到一个称为 Dijkstra 算法,计算从 s 出发到图中各顶点的最短路径。

Dijkstra 算法的伪代码描述如下。

```
DIJKSTRA(G,w,s)
for each vertex v∈V[G]
    do d[v]←∞
       π[v]←NIL
d[s]←0
Q←V[G]
while Q≠∅
    do u←EXTRACT-MIN(Q)
       for 每个与 u 相邻的顶点 v
         do if d[v]>d[u]+w(u,v)
               then d[v]←d[u]+w(u,v)
```

```
11                    π[v]←u
12          FIX(Q)
13 return d and π
```

算法 9-12　解决单源最短路径问题的 DIJKSTRA 算法

按照本问题的贪婪算法性质，应当维护一个顶点集合 S，此集合包含所有已确定从 s 出发的最短距离的顶点。由于把 $V-S$ 的顶点都存放于最小优先队列 Q 中，凡是已出队的顶点，都在 S 中，所以 S 可以不显式地加以维护而被忽略。

算法对每个顶点 v 维护两个属性：表示从 s 经过 S 中的顶点到达 v 的最短路径的距离 $d[v]$和 v 在这段最短路径中的前序顶点 $\pi[v]$。第 1～4 行对记录各顶点的这两个属性的数组 d 和 π 进行初始化。

第 5 行是将优先队列 Q 初始化为图 G 的顶点集 V。

第 6～12 行是按照贪婪选择性质解决此问题：每次在 $V-S(=Q)$中选取 d 值最小的顶点 u 加入到 S 中(从 Q 中出队)，调整所有从 u 出发且尚留驻在 $V-S(=Q)$中的相邻顶点 v 的 d 值及 π 值，并维护 Q 的堆性质。

算法运行于一个有向带权图的实例如图 9-13 所示。

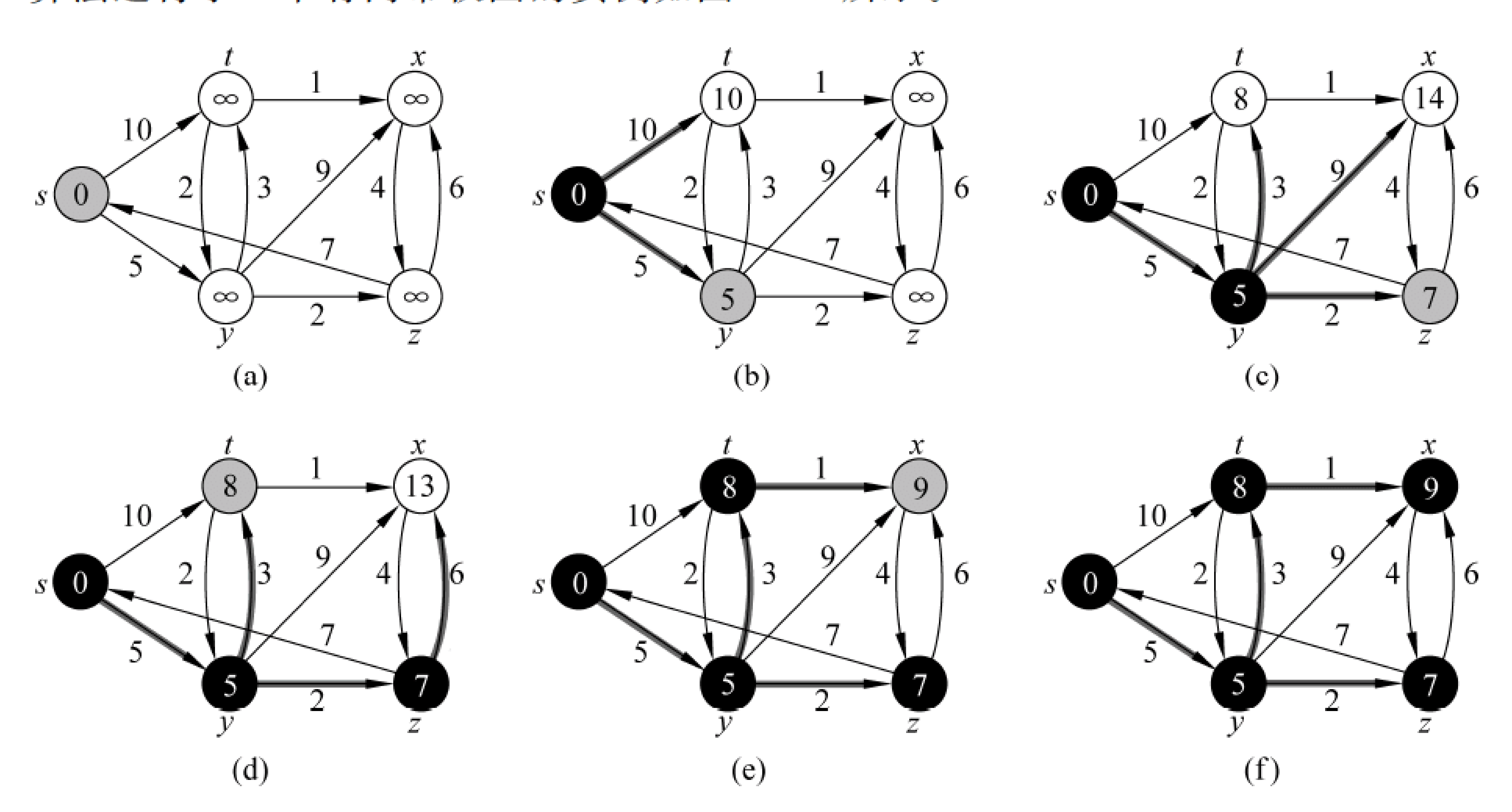


图 9-13　用 Dijkstra 算法计算从 s 出发到图中各顶点的最短路径的示例

5. 算法的运行时间

算法中第 1～3 行的 **for** 循环耗时 $\Theta(n)$，第 6～12 行的 **while** 循环重复 $\Theta(n)$次，每次重复第 7 行的优先队列出队操作耗时 $\Theta(\lg n)$，第 8～11 行的 **for** 循环耗时 $\Theta(n)$，第 12 行对优先队列进行堆性质维护操作将耗时 $\Theta(n)$。因此，总耗时 $\Theta(n^2)$。

利用算法返回的数组 d 和 π，可以输出图中从源顶点 s 到其余各顶点的最短路径及其距离。

```
PRINT-PATH(π,s,v)
  if v=s
```

```
    then print s
    else if π[v]=NIL
            then print "没有从"s"到"v"的路径"
            else PRINT-PATH(π,s,π[v])
                print v
```

算法 9-13　输出图 G 中从源顶点 s 到顶点 v 的最短路径算法

该算法的时间复杂度显然是 $\Theta(n)$。因为一条简单路径至多包含 n 个不同的顶点,所以至多递归调用 n 次。

9.4.2　程序实现

根据对算法 9-12 的说明,必须准备一个数据结构:动态优先队列。这件事情在 9.4.1 节中都已做好了,代码存储在 DataStructure 目录中的头文件 pqueue.h 和源文件 pqueue.c 中。插入到优先队列中的元素是指向数组 d 的元素的指针,所以,必须有一个通过两层指针比较浮点数的函数,这在 9.4.1 节也做好了,原型声明存储在 Utility 目录中的头文件 compare.h 的函数 dblLess 中。

```
#include "../DataStructure/pqueue.h"
#include "../DataStructure/pair.h"
#include "../Utility/compare.h"
pair dijkstra(double *w,int n,int s){
    int *pi=(int*)malloc(n*sizeof(int)),
        *poped=(int*)malloc(n*sizeof(int)),i,u,v;
    double *d=(double*)malloc(n*sizeof(double));
    PQueue *Q=initPQueue(sizeof(double*),n,dblLess);
    for(i=0;i<n;i++){
        pi[i]=-1;
        poped[i]=0;
        d[i]=DBL_MAX;
    }
    d[s]=0.0;
    for(i=0;i<n;i++){
        double *e;
        e=&d[i];
        enQueue(Q,&e);
    }
  while(!empty(Q)){
        u=(*(double**)deQueue(Q))-d;
        poped[u]=1;
        for(v=0;v<n;v++){
            if(!poped[v])
                if(w[u*n+v]>0.0)
                    if(d[v]>d[u]+w[u*n+v]){
```

```
27                     d[v]=d[u]+w[u * n+v];
28                     pi[v]=u;
29                 }
30         }
31         fix(Q);
32     }
33     pQueueClr(Q);
34     return make_pair(d,pi);
35 }
36 void printPath(int * pi,int s,int i){
37     if(i==s){
38         printf("%d ",i+1);
39         return;
40     }
41     if(pi[i]==-1)
42          printf("no path from %d to %d\n",s+1,i+1);
43     else{
44          printPath(pi,s,pi[i]);
45         printf("%d ",i+1);
46     }
47 }
```

程序 9-10　实现算法 9-12 和算法 9-13 的 C 源代码

对程序 9-10 的说明如下。

(1) 第 4～35 行定义的函数 dijkstra 的 3 个参数分别为表示图的权矩阵 w(这实际上已经把图的权函数也表示出来了)、权矩阵的列数 n、源顶点 s。该函数的返回值类型是 8.3.3 节中定义的 pair,其中封装了计算所得的数组 d 和 pi。

(2) 第 5～7 行声明了 3 个数组：记录生成树中每个结点的父结点的数组 pi,跟踪顶点是否在队列中的数组 poped 和跟踪各顶点的 d 属性的数组 d。第 9～13 行的 **for** 循环完成对这 3 个数组的初始化工作。第 14 行将源顶点 s 的 d 值置为 0,作为优先队列的首元素。所有这些代码实现 DIJKSTRA 过程中的第 1～4 行的操作。

(3) 第 15～19 行代码实现 DIJKSTRA 过程中第 5 行的操作 $Q \leftarrow V[G]$。要注意的是,与程序 9-8 的代码解析中的说明相仿,第 16 行声明的变量 e 是用来作为将指向数组 d 的元素指针加入到队列中的中转变量的。

(4) 第 20～32 的代码实现 DIJKSTRA 过程中第 6～12 行的操作。注意第 8 行对动态优先队列 Q 的初始化操作时,传递在程序 9-7 中定义的函数 dblLess。

其次,第 21 行调用函数 deQueue 将 Q 的队首元素出队,解析出其中存储的指向数组 d 的元素的地址,与 d 的首地址之差,恰好是该元素在 d 中的下标值,赋值给表示图中对应顶点的 u。第 22 行将出队的顶点 u 做出队标志(poped[u]置为 1),这个标志是第 24 行所要检测的。

此外,请注意第 25～27 行对由参数传递进来的数组 w 的元素 w[u * n+v]的访问相当于若 w 是一个二维数组对元素 w[u][v]的访问。第 34 行将数组 d 和 pi 封装到 pair 对象

中,并将其返回。

(5)第 36～47 行定义的递归函数 printPath 实现的是算法 9-13,它负责利用数组 pi 打印出从源顶点 s 到顶点 i 的最短路径。代码与算法的伪代码几乎一一对应,此不赘述。

为便于重用,程序 9-10 存储在文件夹 greedy 中的源文件 dijkstra. c 中,函数的原型声明存储在头文件 dijkstra. h 中。

9.4.3 应用——西气东送

Pipe Laying Problem

Description

The West-to-East natural gas transmission project is another important investment project that is next to the Three Gorges Project. In planning The West-to-East natural gas transmission project, it started in Xinjiang Tarim Lunnan oil and gas field in the west, travel eastward through several large and medium-sized cities , such as Korla, Turpan, Shanshan, Hami, Liuyuan, Jiuquan, Zhangye, Wuwei, Lanzhou, Dingxi, Xian, Luoyang, Xinyang, Hefei, Nanjing, Changzhou etc.. The termination is Shanghai. It crossed through 7 provinces or Autonomous regions; they are Gansu, Ningxia, Shanxi, Shanxi, Henan, Anhui, and Jiangsu. In this project, a lot of pipelines of gas transmission are needed to be laid. Now we assume that the pipe can only be laid with straight line way, and some stumbling blocks (such as building etc.) can't be cut through by the pipeline during the process of pipe laying. In other words, if some stumbling blocks are on the straight segment which is between any two gas transmission stations, then the pipeline can not be laid between the two stations (The error margin is less than 10^{-5}). Moreover, for the geology structure is different and other reasons, the budget per unit distance of the pipe laying between any two stations is different. The situation of pipe laying as shown in Figure 9-14.

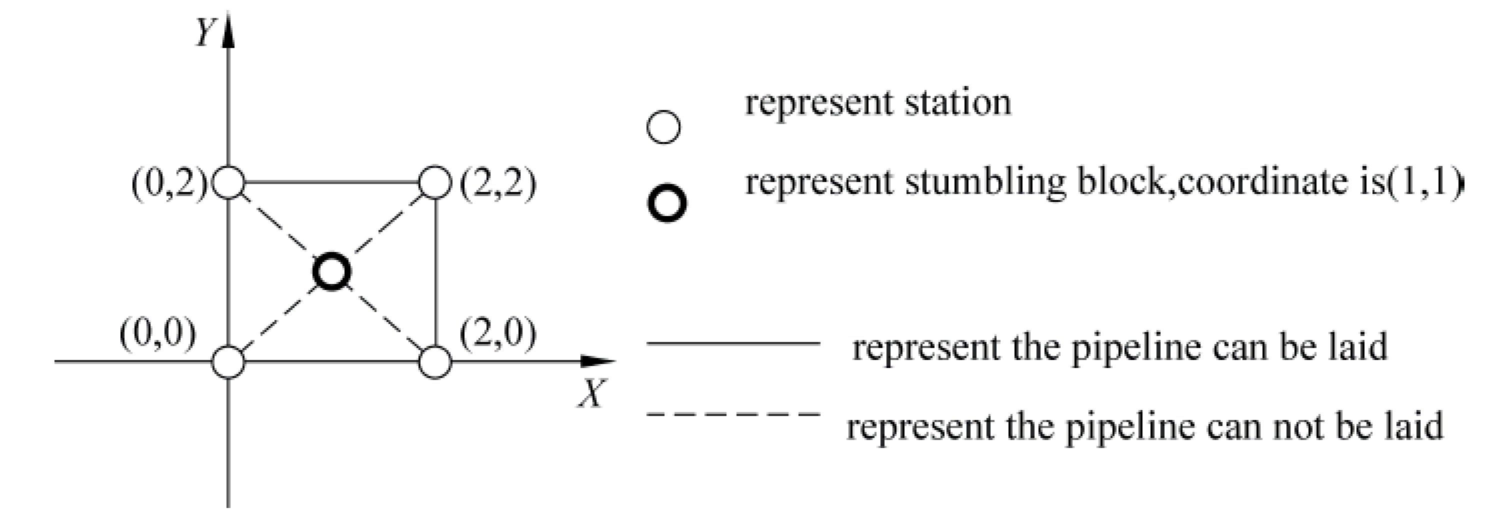


图 9-14　管道铺设的障碍

Please calculate the lowest budget of the pipe laying between two stations. For simplicity, the station and the stumbling block are regarded as a point, that is to say, the width of the stations and the stumbling blocks is ignored.

Input

The first line of the input is a positive integer T. T is the number of the test cases . For each test case, the first line including a single integer N ($3\leqslant N\leqslant 100$) which indicates the number of the stations. Then N lines follow, each has 2 integers separated by blank, which represents the x and y ($-5000\leqslant x\leqslant 5000$, $-5000\leqslant y\leqslant 5000$) coordinate of one station. After that there's a line including a single integer M ($0\leqslant M\leqslant 50$) which indicates the number of the stumbling blocks. Then M lines follow, each has 2 integers separated by space, which represents the x and y ($-5000\leqslant x\leqslant 5000$, $-5000\leqslant y\leqslant 5000$) coordinate of one stumbling block. Then $1/2 * N * (N-1)$ line followed, each line has one not negative real number which indicates the pipeline's budget per unit distance, for example first point to second point, first point to third point , …, first point to N point, second point to third point, …, $N-1$ point to N point. The last line includes two integers which represent input serial numbers.

Note: About the pipeline's budget per unit distance, we don't consider about the stumbling block.

Output

For each test data, you should print one line include one integer which has been rounded. For example: 3.33 is rounded to integer 3, 3.50 is rounded to integer 4, 3.76 is rounded to integer 4, etc. The integer indicates the lowest budget.

Sample Input

```
1
4
0 0
0 2
2 2
2 0
1
1 1
1.2
2.5
3.2
2.1
2
2.7
2 4
```

Sample Output

```
9
```

1. 问题描述与分析

西气东送工程中有 N 个气站，由于气站所处地理位置等因素，气站间铺设管道的代价

各不相同。假定任意两气站间的管道只能沿直线铺设,且若连接两气站的直线段内存在障碍物,则不能铺设管道。需要计算两个指定气站间最小的管道铺设代价。问题可形式化为如下。

输入:N 个气站的坐标 $A=\{(x_1,y_1),\cdots,(x_N,y_N)\}$,$M$ 个障碍物的坐标 $B=\{(x_1',y_1'),\cdots,(x_M',y_M')\}$,气站间铺设管道的单位代价 $p=\{p_1,p_2,\cdots,p_{N\times(N-1)/2}\}$起点气站 s,终点气站 t。

输出:从 s 到 t 的铺设管道的最小代价。

仔细阅读本题,N 个气站可以视为一个图的 N 个顶点。由顶点间铺设输油管道的单位成本和顶点间的距离(通过给出的顶点的坐标计算可得)可计算出顶点间的铺设输油管道的实际成本,这可以视为图中顶点间的边及其权值。这样,本问题就转换成在一个无向带权图中计算出从源点(起点气站)到目标点(终点气站)的最短路径。不过这个图需要从给定的原图做必要的修改得来:删掉含有障碍的边。可以如下描述解决此问题的算法。

2. 算法描述

```
PIPE-LAYING(A,B,p,s,t)
1 w←MAKR-GRAPH(A,p)
2 FIX-GRAPH(G,B)                        ▷w 是图 G 的权矩阵
3 d←DIJKSTRA(G,w,s)
4 return d[t]
```

算法 9-14 计算从 $A[s]$到 $A[t]$的最小输油通道费用的过程

其中,第 1 行调用过程将其站坐标数组 A 和各气站间铺设有关的费用数组 p 转换成一个无向带权图,返回该图的权矩阵 w(元素 $w[u,v]$表示直接铺设气站 u 到气站 v 的输油通道所需费用)。该过程的伪代码列为算法 9-15。第 2 行调用 FIX-GRAPH 过程,在图中删掉含有障碍(数组 B 中元素)的边,得到修改后的图的权矩阵 w。该过程的伪代码列为算法 9-16。第 3 行调用算法 9-12 的 DIJKSTRA 过程,计算图中从 s 出发所有到可达顶点的最短路径(即从气站 s 到气站 t 的最小铺设有关费用),返回每条最短路径的长度构成的数组 d。第 4 行返回 $d[t]$,即 s 到 t 的最短路径长度,亦为本题所求。

```
MAKE-GRAPH(A,p)
1 N←length[A]
2 k←1
3 for u←1 to N-1
4     do w[u,u]←0
5        for v←u+1 to N
6           do w[u,v]←w[v,u]←(A[u],A[v]间的距离)×p[k]
7              k←k+1
8 return w
```

算法 9-15 将气站坐标数据转换成无向带权图

注意:数组 p 中的元素 $p[1]$、$p[2]$、…、$p[N(N-1)/2]$表示顶点 1 到顶点 2,顶点 1 到顶点 3,…,顶点 $N-1$ 到顶点 N 的铺设管道的单位费用。第 3~7 行的两重嵌套的 **for** 循环,循环体恰好重复 $N(N-1)/2$。由于无向图的权矩阵是对称的,所以第 6 行对 $w[u,v]$和

$w[v,u]$赋予相同的值。

```
FIX-GRAPH(G,B)
1 M←length[B]
2 for i←1 to M
3     do for each e∈E[G]
4        do if B_i 含于 e
5              then 在 E[G]中删除 e
```

算法 9-16　在图中删除含有障碍的边

注意：数组 B 中元素为障碍物坐标。第 2 行和第 3 行的 **for** 循环对每一个障碍物 $B[i]$，寻找图中含有 $B[i]$的边加以删除。

3. 程序实现

1）构造无向带权图

实现将输入中的关于气站坐标的数据转换成一个无向带权图的过程。

```
1 double *makeGraph(Point *A,double *p,int n){
2     int i,j,k=0;
3     double *w,d;
4     assert(w=(double*)calloc(n*n,sizeof(double)));
5     for(i=0;i<n;i++)
6         for(j=i+1;j<n;j++){
7             d=sqrt((A[i].x-A[j].x)*(A[i].x-A[j].x)+(A[i].y-A[j].y)*(A[i].y-A[j].y));
8             w[i*n+j]=w[j*n+i]=d*p[k++];
9         }
10     return w;
11 }
```

程序 9-11　实现算法 9-15，将坐标数据转换成无向带权图的函数

用本书第 5 章程序 5-1 定义的数据类型 Point 表示气站坐标及障碍物坐标。实现函数的代码结构接近算法的伪代码。需要注意如下两个问题。

(1) 第 7 行调用库函数 sqrt 计算点 A[i]、A[j]之间的距离。

(2) 用一维数组按行优先原则存储图的权矩阵 w。

2）修改图

对构造好的无向图，需要将经过障碍的边从中删除。这可以对每一个障碍物 $B[i]$依次检测图中每一条边是否含有 $B[i]$来决定是否删除该条边。事实上，并不需要对图中所有的边做检测。设边(u,v)及点 p，它们的坐标为 $u(x_u,y_u)$、$v(x_v,y_v)$、$p(x_p,y_p)$。则(u,v)含有 p 必满足 $\min\{x_u,x_v\}\leqslant x_p\leqslant\max\{x_u,x_v\}$且叉积 $(u-p)\times(p-v)=0$。可以对 A 中的点按 x 坐标的升序排序。对每一个 $B[i]$，找出 A 中横坐标大于 $B[i]$的横坐标的最小者，记为 $bound_i$，只需要考虑边$(A[1],A[bound_i])$，…，$(A[1],A[N])$，$(A[2],A[bound_i])$，…，$(A[2],A[N])$，…，$(A[bound_i-1],A[bound_i])$，…，$(A[bound_i-1],A[N])$是否含有 $B[i]$。

```
1 int pLess(Point **a,Point **b){
2     if((*a)->x>(*b)->x)
3         return 1;
4     if((*a)->x<(*b)->x)
5         return -1;
6     if((*a)->y>(*b)->y)
7         return 1;
8     if((*a)->y<(*b)->y)
9         return -1;
10    return 0;
11 }
12 void fixGraph(double *w,Point *A,Point *B,int n,int m){
13    Point **a;
14    int *bounds=(int *)malloc(m*sizeof(int));
15    int i,j,index;
16    assert(a=(Point**)malloc(n*sizeof(Point*)));
17    for(i=0;i<n;i++)
18        a[i]=A+i;
19    qsort(a,n,sizeof(Point*),pLess);
20    for(i=0;i<m;i++){
21        index=0;
22        while((a[index])->x<B[i].x)
23            index++;
24        bounds[i]=index;
25    }
26    for(i=0;i<m;i++){
27        int u,v;
28        for(u=0;u<bounds[i];u++)
29            for(v=bounds[i];v<n;v++)
30                if(!direction(a[u],&B[i],a[v])){
31                    int p=a[u]-A,q=a[v]-A;
32                    w[p*n+q]=w[q*n+p]=DBL_MAX;
33                }
34    }
35    free(a);free(bounds);
36 }
```

程序 9-12　实现算法 9-16,删去图中含有障碍物的边的函数

对程序 9-12 的说明如下。

(1) 第 12～36 行定义的函数 fixGraph 实现算法 9-16,删除图中含有障碍物的边。用权矩阵 w 表示图,但 w[u,v]虽然反映了 u 号气站到 v 号气站直接铺设管道的费用,但并不包含 u、v 的坐标信息。所以,参数除了矩阵 w 和障碍物坐标数组 B 外,还需要传递各顶点坐标信息的数组 A。参数 n 和 m 分别表示顶点数(气站数)和障碍物数。

(2) 为了加快在图中查找含有障碍物 B[i]的边,要对数组 A 排序。然而,矩阵 w 是按

原 A 中气站的顺序构造的，一旦 A 重排后，要对两个气站对应的一条边反查 w 中的元素会带来困难。所以第 13 行、第 16 行定义了一个数组 a，第 17 行和第 18 行将数组 A 中元素的地址存放到 a 中。第 19 行调用库函数 qsort 对 a 进行排序。注意，传递给 qsort 的第 4 个参数是指向第 1～11 行定义的比较，由 2 个两重 Point 指针指向气站坐标数据的“大小”。这样，a 中的元素（A 的元素的地址）得到了重排，而 A 的元素保持原来的顺序。

（3）第 20～25 行的 **for** 循环计算每一个 B[i]的“界限”顶点：该顶点左边的顶点横坐标小于 B[i]的横坐标，该顶点右边的顶点（包括该顶点）的横坐标大于 B[i]的横坐标。记该点为 bounds[i]。第 26～34 行的 3 重嵌套循环对每一个障碍物 B[i]查找包含它的边并加以删除。其中，第 28～34 行的循环对 B[i]左边的顶点 a[u]及右边的顶点 a[v]构成的边检测B[i]是否含于该条边。第 30 行调用第 5 章程序 5-2 定义的函数 direction，检测 a[u]、B[i]、a[v]是否共线（函数返回值为 0）。若是，第 31 行确定 a[u]、a[v]在 A 中的位置 p、q，第32 行反查w[p，q]和 w[q，p]并置为 DBL_MAX（用此常量代替表示∞），将此边删除。

3）**计算最优管道**

有了上述的准备，现在来实现计算最优管道的计算过程。

```
1 int round(double x){
2     int a=x;
3     if(x-a>=0.5)
4         a++;
5     return a;
6}
7 int pipeLaying(Point *A,int n,Point *B,int m,double *p,int s,int t){
8     pair r;
9     int x;
10    double *w=makeGraph(A,p,n), *d;
11    fixGraph(w,A,B,n,m);
12    r=dijkstra(w,n,s-1);
13    d=(double *)r.first;
14    x=round(d[t-1]);
15    free(w);free(r.second);free(d);
16    return x;
17 }
```

程序 9-13　实现算法 9-14 的 C 函数

第 7～17 行定义的函数 pipeLaying，代码与算法的伪代码结构十分接近。需要注意如下问题。

（1）按题面要求，计算出来的从 s 点到 t 点最小铺设费用要四舍五入为整数。所以，第 1～6 行定义了计算浮点型数据四舍五入为整数的函数 round。

（2）第 12 行调用程序 9-10 定义的函数 dijkstra 对修改过的图的权矩阵 w 及起点 s（由于数组下标从 0 开始编码，所以图的顶点编码也从 0 开始，于是传递给 dijkstra 的第 3 个参数为 s－1）计算单源最短路径。回忆该函数返回 2 个计算结果：s 到每一个定点的最短距

离 d 及各条路径中每个顶点的父结点数组 pi。我们只需要数组 d,以及 d 中 d[t](在这里是 d[t－1])。

调用函数 pipeLaying 解决 Pipe Laying 问题的 main 函数定义连同程序 9-11～程序 9-13 都存储在文件夹chap09/Pipe Laying Problem 中的源文件 PipeLayingProblem. c 中,读者可打开文件研读。

第 10 章　图的搜索算法

图是用来表示现实世界中对象之间关系的数学模型，所以它是信息技术中用来表达各种应用系统的强有力的逻辑模型。随着信息技术应用的不断深化，在模拟人的智力活动的人工智能技术中，表达知识就需要借助图。图的搜索指的是系统地沿图的边访问图中的顶点的过程。一个图搜索算法可能发现关于图的结构的更多信息。很多算法是从搜索输入的图而得到这些结构信息开始的，还有很多图算法是对基本图搜索算法的简单扩展。图的搜索技术是图算法领域的核心，本章运用前 9 章讨论过的算法设计方法来设计有效的图的搜索算法，并探讨几个基本应用问题。

从第 3 章起，我们就讨论过图。对图(有向或无向)$G=<V,E>$(为方便，假定 $V=\{1,2,\cdots,n\}$)，是用图的邻接矩阵 $\mathbf{A}=(a_{ij})_{n\times n}$ 来表示的。其中，$a_{ij}=\begin{cases}1 & (i,j)\in E \\ 0 & (i,j)\notin E\end{cases}$，$(1\leqslant i,j\leqslant n)$。在计算机中，矩阵可以很方便地用一个数组加以表示，这在前几章已有体会。但是从算法的时间效率角度考虑，有时需要另外一种表示图 G 的数据结构，这就是现在需要跟大家介绍的图的**邻接表表示**。

图 $G=<V,E>$ 的邻接表表示是一个由 $|V|$ 个链表组成数组，对每个 $u\in V$，链表 $Adj[u]$ 称为对应顶点 u 的**邻接表**。它包含 G 中所有与 u 相邻的顶点。每个邻接表中顶点通常是按任意顺序存放的。图 10-1(b)是图 10-1(a)中的无向图的一个邻接表表示。类似地，图 10-2(b)是图 10-2(a)中的有向图的一个邻接表表示。

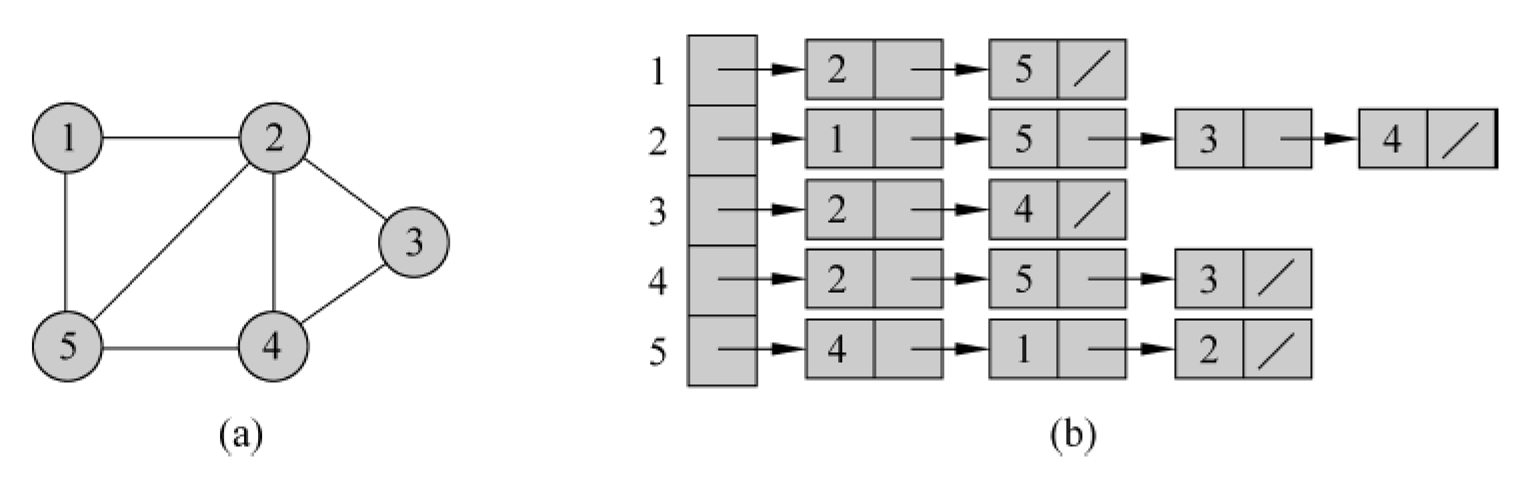


图 10-1　无向图及其邻接表表示

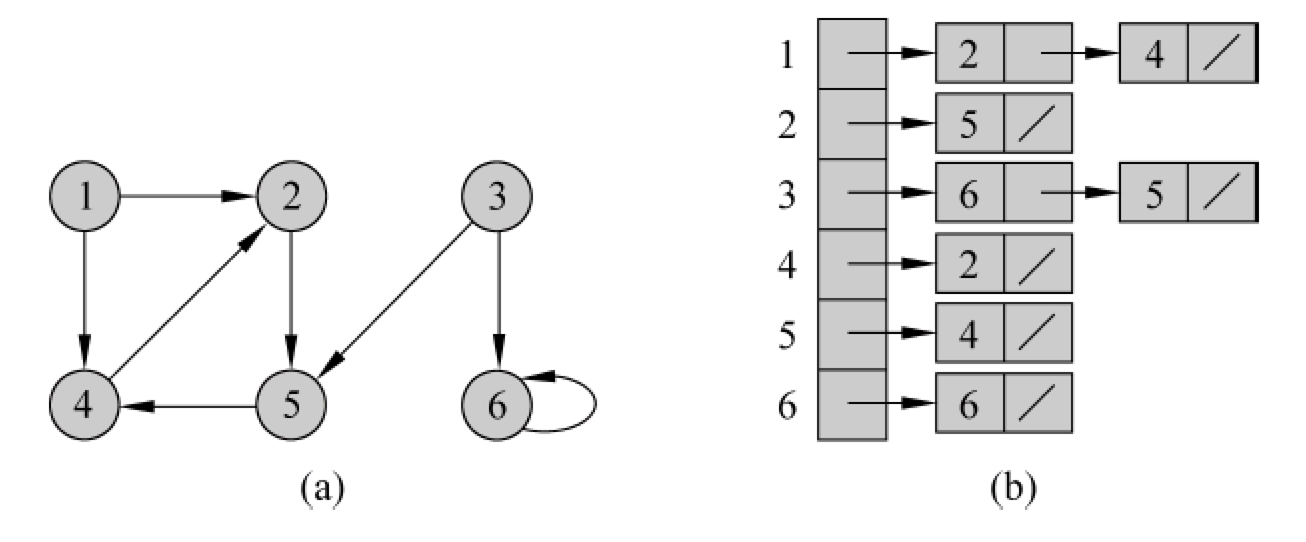


图 10-2　有向图及其邻接表表示

在正式开始讨论图的搜索算法之前，先通过一个例子来了解图的邻接表是如何影响算法的时间效率的。有向图 $G=<V,E>$的**转置**是一个图 $G^T=<V,E^T>$，其中 $E^T=\{(v,u)\in V\times V : (u,v)\in E\}$，即$G^T$ 就是将 G 中的边反向而得。

对邻接矩阵表示的图，算法伪代码如下。

```
TRANSPOSE-DIRECTED-GRAPH(G)
for u←1 to |V| do
  for v←1 to |V| do
    A^T[v,u]←A[u,v]
return G^T
```

算法 10-1 有向图的转置算法(邻接矩阵版本)

其中，A^T 表示 G^T 的邻接矩阵。显然，其运行时间为 $\Theta(|V|^2)$。

对邻接表表示的图，算法伪代码如下。

```
TRANSPOSE-DIRECTED-GRAPH(G)
1 for each u∈V do
2   Adj^T[u]←NIL
3 for each u∈V do
4   for each v∈Adj[u] do
5     INSERT(Adj^T[v],u)
6 return G^T
```

算法 10-2 有向图的转置算法(邻接表版本)

其中，Adj^T 表示 G^T 的邻接表。第 1 行和第 2 行耗时$|V|$，第 3 行和第 4 行耗时$|E|$，所以，总耗时 $\Theta(|V|+|E|)$。

当$|E|$与$|V|$渐近相等时，显然用邻接表表示的图的算法比邻接矩阵表示的图的算法更好。所以，本章中除非特别声明，总是假定图(无论是有向图还是无向图)是用其邻接表表示的。

10.1 深度优先搜索

10.1.1 算法描述与分析

1. 问题的理解与描述

深度优先搜索(Depth First Search，DFS)所遵循的策略，如同其名称所云，是在图中尽可能“更深”地进行搜索。在深度优先搜索中，对最新发现的顶点 v，若此顶点尚有未探索过从其出发的边就进行探索。当 v 的所有边都被探索过，搜索“回溯”到从其出发发现顶点 v 的顶点。此过程继续直至发现所有从源点可达的顶点。在这个搜索过程中，只要在某个已发现的顶点 v 的邻接表中发现一个顶点 u，则将 v 的父亲指针 $\pi[u]$置为 v。这样会形成一棵以源点为根的树——深度优先树。若图中还有未发现的顶点，则以其中之一为新的源点重复搜索，直至所有的顶点都被发现。这样，搜索轨迹 G_π 将形成一片由若干棵深度优先

树构成的森林——深度优先森林。

在搜索过程中，用顶点的颜色来指示顶点的状态。每一个顶点初始时是白色的，搜索一旦被**发现**就变成灰色，当其**完成**时也就是它的邻接表被完全考察过就成为黑色的。为了通过深度优先搜索揭示图的更多信息，在深度优先搜索过程中对每一个顶点 u 跟踪两个时间：发现时间 $d[u]$和完成时间 $f[u]$。$d[u]$记录首次发现(u 由白色变成灰色)时刻，$f[u]$记录完成 v 的邻接表检测(变成黑色)时刻。换句话说，对每个顶点 u，必有

$$d[u] < f[u] \tag{10-1}$$

在时间 $d[u]$前顶点 u 是白色的(WHITE)，在时间 $d[u]$和 $f[u]$之间是灰色的(GRAY)，$f[u]$之后是黑色的(BLACK)。所有这些时间是介于 1 到 2 $|V|$的整数，这是因为对$|V|$个顶点的每一个而言仅发生一次发现事件和一次完成事件。

把图的深度优先搜索问题形式化为如下。

输入：图 $G=<V,E>$。

输出：G 的深优先森林 G_π 以及图中各顶点在搜索过程中的发现时间和完成时间。

2. 算法的伪代码描述

算法 10-3 的伪代码是基本的深度优先算法，它利用一个栈来控制对顶点的访问顺序。“栈”是一种插入和删除操作都在同一端进行的线性表。它具有元素先进后出的特性。输入的图 G 可以是无向图也可以是有向图。变量 $time$ 是一个用来表示时间的变量。

```
DFS(G)
for each vertex u∈V[G]
    do color[u]←WHITE
       π[u]←NIL
time←0
S←∅
for each vertex s∈V[G]
    do if color[s]=WHITE
        then color[s]←GRAY
             d[s]←time←time+1
             PUSH(S,s)
             while S≠∅
               do u←TOP(S)
                  if ∃[1]v∈ Adj[u] and color[v]=WHITE
                     then color[v]←GRAY
                          π[v]←u
                          d[v]←time←time+1
                          PUSH(S,v)
                     else color[u]←BLACK
                          f [u]←time←time+1
                          POP(S)
 return d,f,and π
```

算法 10-3　对图的深度优先搜索算法

① ∃是数理逻辑中的存在量词，意为“存在……”。

过程 DFS 运行如下。

第 1～3 行将所有的顶点着上白色并将它们的 π 域初始化为 NIL。第 4 行将时间计数器置为 0。第 6～19 行的 **for** 循环依次检测 V 中的每个顶点 s，一旦发现一个白色的顶点，就以此顶点为源顶点，以深度优先的方式搜索从 s 可达的所有顶点。该循环的每次重复，顶点 s 变成深度优先森林中的一棵新树的根。第 8 行将 s 着成灰色，第 9 行将时间计数器 $time$ 增加了 1 并作为 s 的发现时间 $d[s]$，第 10 行将 s 加入到栈 S 中。第 11～20 行的 **while** 循环借助栈 S 以深度优先的方式搜索所有从 s 可达的顶点：每次重复寻找与处于栈顶的顶点 u 相邻且未曾发现过的(白色)顶点 v(第 13 行的检测)。若有这样的顶点，则将其改为灰色，记录其发现时间 $a[v]$，父指针 $\pi[v]$，并压入栈 S，这时说深度优先搜索**探索**了边(u,v)。否则，将处于栈顶的顶点 u 着成黑色，记录完成时间 $f[u]$ 将其从栈 S 中弹出。当 DFS 返回时，图中每个顶点 u 就被赋予一个**发现时间** $d[u]$和一个**完成时间** $f[u]$，以及 u 在深度优先森林中的父结点$\pi[u]$。对一个图的 DFS 的过程如图 10-3 所示。

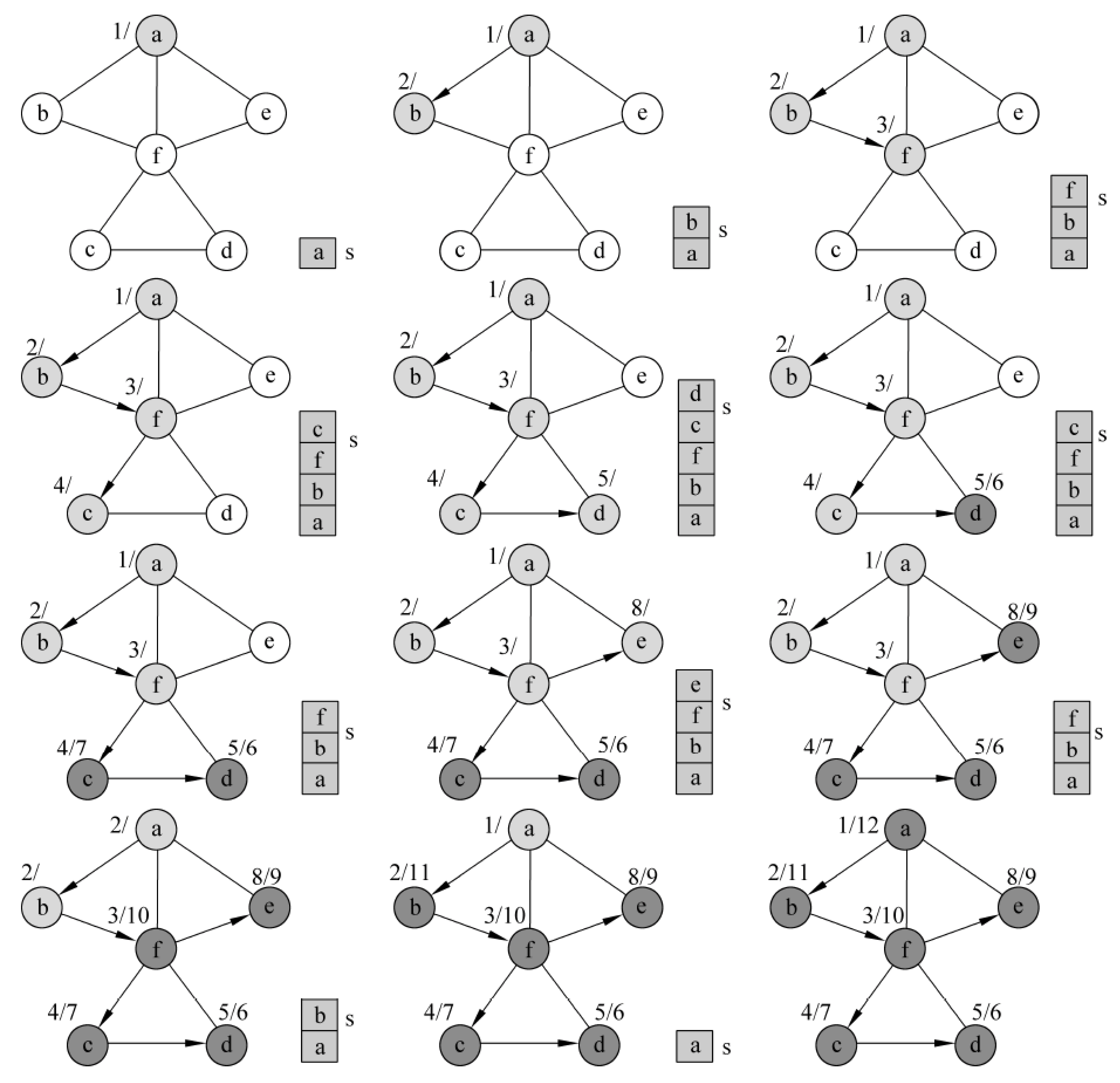


图 10-3 深度优先搜索算法 DFS 施于一个图的过程(各顶点按发现时间/完成时间格式做标记)

注意：深度优先搜索的结果可能依赖于 DFS 的第 6 行所检测的顶点的顺序，以及第 13 行所访问的各相邻顶点的顺序。这些不同的访问顺序并不会在实践中发生什么问题，但

搜索形成的深度优先搜索森林的形态可能有所不同。深度优先搜索的任一结果都可用另一本质上等价的结果替代。

3. 算法的运行时间

DFS 的运行时间如何？第 1 行和第 2 行的循环耗时 $\Theta(V)$。内嵌于第 14～20 行操作对 G 的每条边执行一次，因此耗时

$$\sum_{v \in V} |Adj[v]| = O(E)$$

所以 DFS 的运行时间为 $\Theta(V+E)$。

10.1.2　程序实现

1. 图的邻接表表示

用邻接表来表示图。利用在第 2 章开发的链表 LinkedList（保存在 DataStructure 目录下的头文件 list.h 和源文件 list.c 内）作为每个顶点的邻接表。

```
typedef struct{
    double weight;                                   /* 权 */
    int index;                                       /* 顶点编号 */
} vertex;
typedef struct Graph{
    LinkedList **adj;                                /* 邻接表数组 */
    int n;                                           /* 顶点数 */
} Graph;
int vComp(vertex *a,vertex *b){                     /* 两个结点 a、b 的比较 */
  return a->index-b->index;                          /* 按编号比较大小 */
}
Graph *zeroGraph(int n){                            /* 创建零图 */
  int i;
  Graph *g=(Graph *)malloc(sizeof(Graph));
  g->n=n;                                            /* 确定顶点个数 */
  g->adj=(LinkedList**)malloc(n*sizeof(LinkedList *));
  for(i=0;i<n;i++)                                   /* 为每个顶点创建邻接表 */
    g->adj[i]=createList(sizeof(vertex),vComp);
  return g;
}
void addEdge(Graph *g,int u,int v,double w){        /* 向图中添加边 */
  vertex x={w,v};                                    /* 边(u,v)的权为 w */
  listPushBack(g->adj[u],&x);
}
void graphClear(Graph *g){                          /* 清理图的邻接表存储空间 */
  int i;
  LinkedList *l;
```

```
28  for(i=g->n-1;i>=0;i--){                    /* 清理每个顶点的邻接表 */
29    l=g->adj[i];
30    clrList(l,NULL);
31    free(l);
32  }
33  free(g->adj);                               /* 释放邻接表数组 */
34  g->n=0;
35 }
```

程序 10-1 实现图的邻接表表示的 C 源代码

对程序 10-1 的说明如下。

(1) 第 1~4 行定义了顶点邻接表中结点的数据类型 vertex。它包含两个属性：表示顶点编号的 index 及表示与宿主顶点构成的边的权值 w。在表示无权图时，w 的值均为 1。第 5~8 行定义了表示图的邻接表的数据类型 Graph。它有 2 个属性：表示邻接表数组的 adj 和表示顶点个数的 n。

(2) 第 12~20 行定义的函数 zeroGraph 创建一个具有 n 个顶点的**零图**①。其中，第 15 行确定图 g 有 n 个顶点。第 17 行和第 18 行的 **for** 循环为每个顶点创建 LinkedList 型的邻接表。注意，调用函数 createList 创建链表时传递的第 1 个参数表示链表中每个结点存放 vertex 类型的数据，第 2 个参数传递的是用于在链表中查找结点时所需的比较结点的函数指针 vComp，该函数定义在第 9~11 行。

第 21~24 行定义的函数 addEdge 在由参数 g 指引的图中，插入由参数 u 和 v 决定的边。参数 w 表示边(u,v)的权。第 22 行用 w 和 v 生成一个结点 x，第 23 行将 x 插入到链表 g－>adj[u]中。

(3) 由于图的邻接表中含有链表，在表示图的数据废弃不用前需清理所占用的动态空间。第 25~35 行定义的函数 graphClear 就是用来清理由参数 g 表示的图的邻接表的存储空间的。其中，第 28~32 行的 **for** 循环调用函数 clrList 清理链表 g－>adj[i]的存储空间。第 33 行清理顶点数组 adj 的存储空间。

为便于代码重用，程序 10-1 中的数据类型定义及函数声明存储在文件夹 graph 中的头文件 graph.h 中，函数定义存储在同一文件夹中的源文件 graph.c 中。

2. 实现 DFS

在实现了图的邻接表表示后，下面实现计算图的深度优先搜索的 DFS 过程。

```
typedef enum {WHITE,GRAY,BLACK} Color;
pair dfs(Graph *g){
    pair *df=(pair *)malloc(sizeof(pair));
    int u,s,n=g->n,v,m=n,time=0,
        *pi=(int *)malloc(n*sizeof(int)),      /* 深度优先森林结点的父结点数组 */
        *d=(int *)malloc(n*sizeof(int)),       /* 发现时间数组 */
        *f=(int *)malloc(n*sizeof(int));       /* 完成时间数组 */
```

① 简单地说，零图就是只有顶点，没有边的图，即 $G=\langle V,E\rangle$，$V=\{1,2,\cdots,n\}$，$E=\emptyset$。

```
8    Stack *S=createStack(sizeof(int));                    /*栈*/
9    Color *color=(Color*)malloc(n*sizeof(Color));  /*顶点颜色数组*/
10   ListNode **pos=(ListNode**)malloc(n*sizeof(ListNode*));
11   for(u=0;u<n;u++){
12       pi[u]=-1;
13       color[u]=WHITE;
14       pos[u]=g->adj[u]->nil->next;
15   }
16   for(s=0;s<n;s++){
17       if(color[s]==WHITE){
18           color[s]=GRAY;
19           d[s]=++time;
20           push(S,&s);
21           while(!stackEmpty(S)){
22             ListNode *p;
23             u=*(int*)(S->top->key);
24               p=pos[u];
25               v=(p!=g->adj[u]->nil)? ((vertex*)(p->key))->index:-1;
26               while(p!=g->adj[u]->nil&&color[v]!=WHITE){
27                   p=p->next;
28                   v=(p!=g->adj[u]->nil)? ((vertex*)(p->key))->index:-1;
29               }
30               pos[u]=p;
31               if(pos[u]!=g->adj[u]->nil){
32                   color[v]=GRAY;
33                   d[v]=++time;
34                   pi[v]=u;
35                   push(S,&v);
36               }else{
37                   color[u]=BLACK;
38                   f[u]=++time;
39                   pop(S);
40               }
41           }
42       }
43   }
44   *df=make_pair(d,f);
45   free(pos);free(color);
46   return make_pair(pi,df);
47 }
```

程序 10-2　实现图的深度优先搜索算法 10-3 的 C 源代码

对程序 10-2 的说明如下。

(1) 第 1 行定义了表示顶点颜色的枚举数据类型 Color，以增加代码的可读性。与算法过程一样，函数 dfs 仅有一个表示图的参数。函数需要返回 3 个数组，表示深度优先森林的

pi、各顶点发现时间 d 和各顶点完成时间 f。设法把 d 和 f 整合在一个 pair 对象中(pair 是在第 8 章的 8.3.4 节中开发的结构体类型),再把 pi 和整合了 d、f 的 pair 对象整合在另一个 pair 对象中返回。

(2) 第 5~7 行声明的 3 个数组 pi、d、f,对应于算法中的数组 π、d、f。第 9 行声明了一个与算法中同名的顶点颜色数组 color。第 8 行声明的是栈类型 Stack 的对象 S,对应于算法中的同名栈。第 11~15 行对应算法 10-3 中的第 1~5 行的初始化工作。

(3) 为实现算法中第 13 行 **if** $\exists v \in Adj[u]$ and $color[v]$=WHITE 的检测,本质上,这需要在顶点 u 的邻接表 $Adj[u]$(这是一个链表)中搜索,直到找到一个白色顶点或发现整个表中不存在白色顶点为止。所以,可以设置一个链表指针 p,初始值为指向 $Adj[u]$的首结点,然后让其在 $Adj[u]$中扫描,直到发现白色顶点或遇到链表尾为止。如果 p 是在 **while** 循环内被初始化为 $Adj[u]$的首结点,而 u 可能会多次出现在栈顶,这样就会使得曾经访问过与 u 邻接的顶点可能会重复被访问,这不符合算法中对每一条边仅访问一次的描述。为避免这样的情形,可以事先为每一个顶点设置一个指向其邻接表的指针 pos[u],初始化为指向 $Adj[u]$的首结点。**while** 的每次重复中对链表 $Adj[u]$的扫描局限于 pos[u]到链表尾的范围内,而不再重新检测 $Adj[u]$首结点到 pos[u]之间的那些已经访问过的非白色顶点。程序中第 10 行声明了一个以链表结点指针 ListNode * 为元素类型的数组 pos,第 14 行将它们初始化为指向每个顶点 u 的邻接表的首元素。pos[u]用来跟踪顶点 u 的邻接表中尚未扫描过的第一个元素位置。这样,就可保证在 DFS 过程中,每一条边有且仅有一次被访问。

(4) 第 16~43 行的 **for** 循环对应算法 10-3 中的第 6~20 行构造图的深度优先森林的操作。其中第 24~29 行实现算法中的第 13 行寻找顶点 u 的相邻白色顶点:第 26~29 行的 **while** 循环在 u 的邻接表中依次查找白色顶点,注意循环的条件是 p!=g->adj[u]->nil &&color[v]!=WHITE,其中的 p 是在第 24 中声明并初始化为 pos[u]的一个链表迭代器,v 是在第 25 行被初始化为 p 所指向的与 u 相邻的顶点。于是,作为逻辑与的第一个条件 p!=g->adj[u]->nil 指的是尚在 u 的邻接表中扫描,而第二个条件 color[v]!=WHITE 检测的是表中当前元素对应的顶点是否为白色。该循环停止时或 p 指向了一个白色顶点(第 30 行将此位置记录到 pos[u]中),这意味着 p!=g->adj[u]->nil,第 31 行检测到此条件将引发第 32~35 行的操作,它们对应算法 10-3 中的第 14~17 行的操作。否则,意味着顶点 u 的邻接表扫描结束仍未发现白色顶点,这将导致执行对应于算法中第 18~20 行的操作的第 37~39 行。

(5) 第 46 行将数组 pi、d 和 f 封装到两个嵌套的 pair 对象中返回。为便于代码重用,把程序存储在文件夹 graph 中的头文件 dfs.h(函数的原型声明)和源文件 dfs.c 中。

10.1.3 有向无圈图的拓扑排序

1. 深度优先搜索的性质

深度优先搜索最基本的性质也许就是搜索轨迹构成若干棵树的**深度优先森林**。首先看到,图中的每个顶点有且仅有一次被发现和完成。所以,每个非源顶点的父结点是唯一的

(源顶点没有父结点)，也就是说，从一个源顶点起，搜索过程将所有从源顶点可达的顶点构成一棵搜索树。但图中一个顶点 u 不必从另一个顶点 s 可达，所以在以 s 为源顶点进行深度优先搜索过程中，u 不必成为搜索树中的结点。因此，u 可能成为另一个源顶点，搜索过程将构成另一棵搜索树。

深度优先搜索的另一个重要性质是发现时间和完成时间具有**括号结构**。如果把顶点 u 的发现时间表示成左括号"(u"，并将其完成时间表示为"u)"，则发现和完成的历史将构成一个在嵌套括号意义上的良好结构表达式。例如，图 10-4(a)的深度优先搜索对应的加括号展示在图 10-4(b)中。也就是说，图(有向或无向)G 的深度优先森林中顶点 v 为顶点 u 的后代当且仅当 $d[u]<d[v]<f[v]<f[u]$。这从算法的执行也可见：父亲先于孩子进栈(这意味着 $d[u]<d[v]$)，而后于孩子出栈(这意味着 $f[v]<f[u]$)。再由式(10-1)可得 $d[u]<d[v]<f[v]<f[u]$。

深度优先搜索的另一个有趣的性质是它可以用来对作为输入的图 $G=<V,E>$ 的边进行分类。对图的边的这一分类可用来获取有关图的重要信息。例如，一个有向图是无圈的当且仅当一个深度优先搜索将得出无"回"边判断。

可以用由对图 G 做深度优先搜索而产生的深度优先森林 G_π 来定义 4 种类型的边。

(1) **树枝边**是深度优先森林 G_π 中的边。若 v 是在探索边(u,v)时首次被发现，则边(u,v)是一条树枝边。

(2) **回边**是那些从顶点 u 连接到其在深度优先森林中的前辈 v 的边(u,v)。自循环边视为回边。

(3) **进边**是那些从顶点 u 连接到其在深度优先森林中的后代 v 但不是树枝边的边(u,v)。

(4) **跨边**是所有其他的边。它们可以是同一棵深度优先树中两个顶点间的边，但其中的任一个不是另一个的前辈，或它们连接两棵不同的深度优先树。

图 10-4 中，各条边都加了表示它们类型的标识。

DFS 算法可以修改成能对各条边在遇到它们时进行分类。关键的思想是每一条边(u,v)在首次被探索时可以根据顶点 v 的颜色来分类(但是进边和跨边不能区分)。

(1) 白色(WHITE)意味着一条树枝边。

(2) 灰色(GRAY)意味着一条回边。

(3) 黑色(BLACK)意味着一条进边或跨边。

第一种情形由算法可得。对于第二种情形，注意各灰色顶点总是形成一条对应于栈中的后代线性链；灰色顶点数比最近发现的顶点在深度优先森林中的深度还多一个。探索总是从最深的灰色顶点开始，所以达到的另一个灰色顶点的边就到达一个祖先。

图 10-4(a)为一个有向图的深度优先搜索结果。各顶点都标有发现时间/完成时间且各条边的类型。图 10-4(b)每个顶点的发现时间和完成时间构成的区间对应于所展示的加括号。每一个矩形跨越对应顶点的由发现时间和完成时间构成的区间。树枝边也得以展示。若两个区间相交，则一个必嵌套在另一个之中，且对应于较小区间的顶点是对应于较大区间顶点的后代。图 10-4(c)重画图 10-4(a)中部分按树枝边和进边自上而下，而所有的由后代到前辈的回边是自底向上的。

值得注意的是，在一个无向图中，因为(u,v)和(v,u)实际上是一条边。所以在对无向

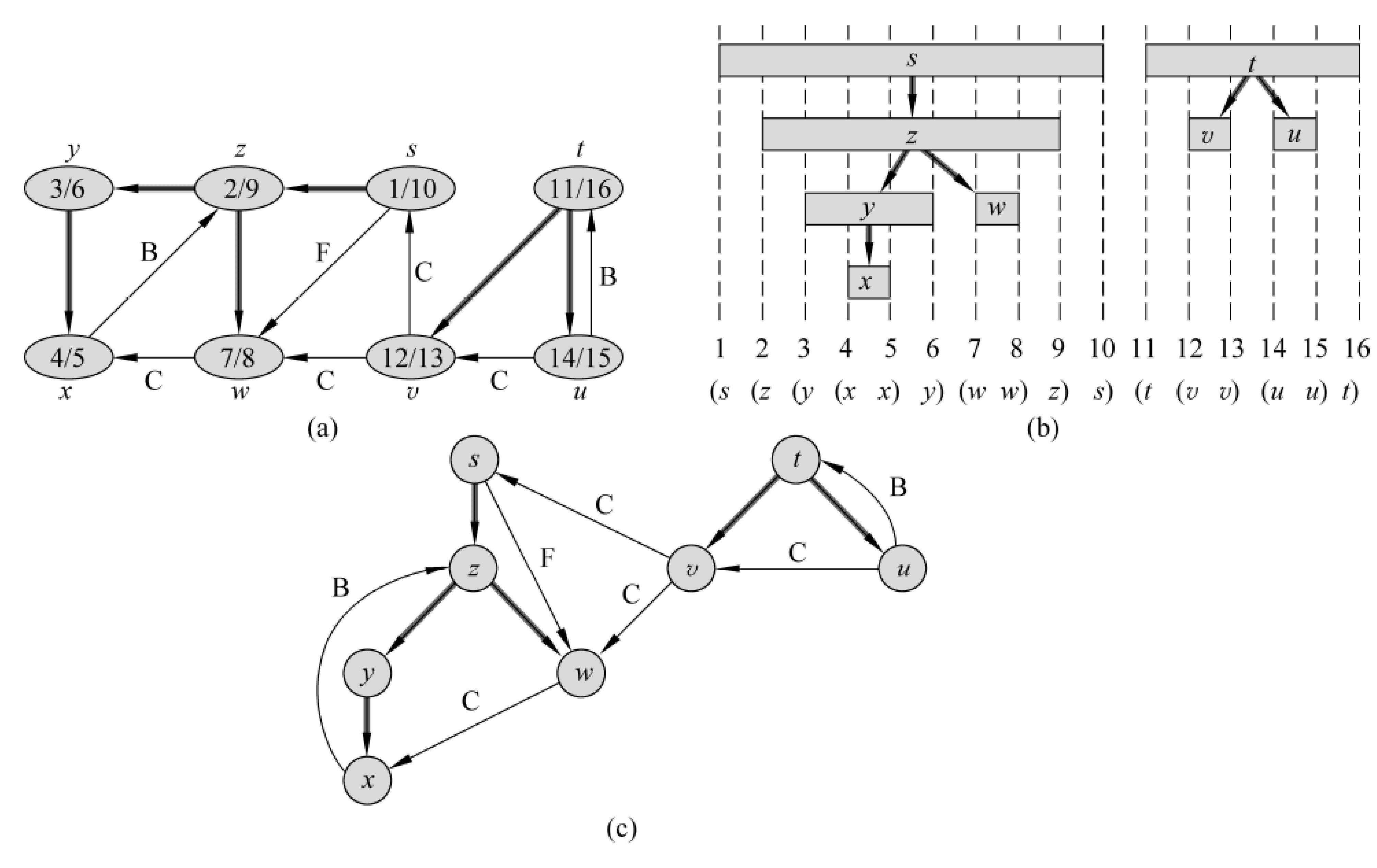


图 10-4 深度优先搜索

图 G 的深度优先搜索中，G 的每一条边或是树枝边或是回边。为说明这一点，设 (u,v) 是 G 的任一条边，不失一般性，假定 $d[u]<d[v]$。于是，v 必在完成 u 之前被发现，这是因为 v 在 u 的邻接表中。若边 (u,v) 先从 u 到 v 被探索到，则 (u,v) 变成树枝边。若 (u,v) 先从 v 到 u 被探索到，则 (u,v) 成为回边，因为 u 在边被首次探索到时还是灰色的。

2. 有向无圈图的拓扑排序

1) 问题理解与描述

由于回边意味着图中存在一个圈，所以得到有向图 G 是无圈的充分必要条件是 G 的一次深度优先搜索不产生回边。这样，对 DFS 稍加修改就可使其能对传递给它的有向图判断是否为有向无圈图(Directed Acyclic Graph，DAG)：设置一个标志 *acyclicity*，初始时置为 true，搜索过程中若遇到回边，则将其置为 false。

用此性质来解决一个有趣的问题：DAG 的**拓扑排序**。一个有向无圈图 $G=\langle V,E\rangle$ 的拓扑排序是其所有顶点的一个线性排列，使得若边 (u,v) 包含在 G 中，则 u 在排列中必出现在 v 前(若图不是无圈的，则不可能有此线性排列)。一个图的拓扑排序可被视为将图的所有顶点水平排列时，所有的有向边从左指向右。因此，拓扑排序与第 1 章和第 2 章中研究的那种“排序”是不同的。把有向图的拓扑排序问题形式化为如下。

输入：有向图 G。

输出：若 G 是 DAG，输出 G 的各顶点的一个拓扑排序，否则输出出错信息。

有向无圈图 DAG 在很多应用中用来说明事件间的先后顺序。图 10-5 给出了发生在某计算机系安排各门课程的教学时的一个例子。由于必须在学习一些课程之前学完一些课程(例如，必须在学习普通物理前学完高等数学)，而对另一些课程却可按任意顺序进行(例如，程序设计与政治经济学等)。在图 10-5(a)中的有向无圈图中的一条有向边 (u,v) 意味

着课程 u 必须在 v 之前学完。所以,此有向无圈图的一个拓扑排序给出了一个教学顺序。图 10-5(b)展示了对此有向无环路图作拓扑排序后水平排列的各个顶点使得所有有向边都是从左指向右。

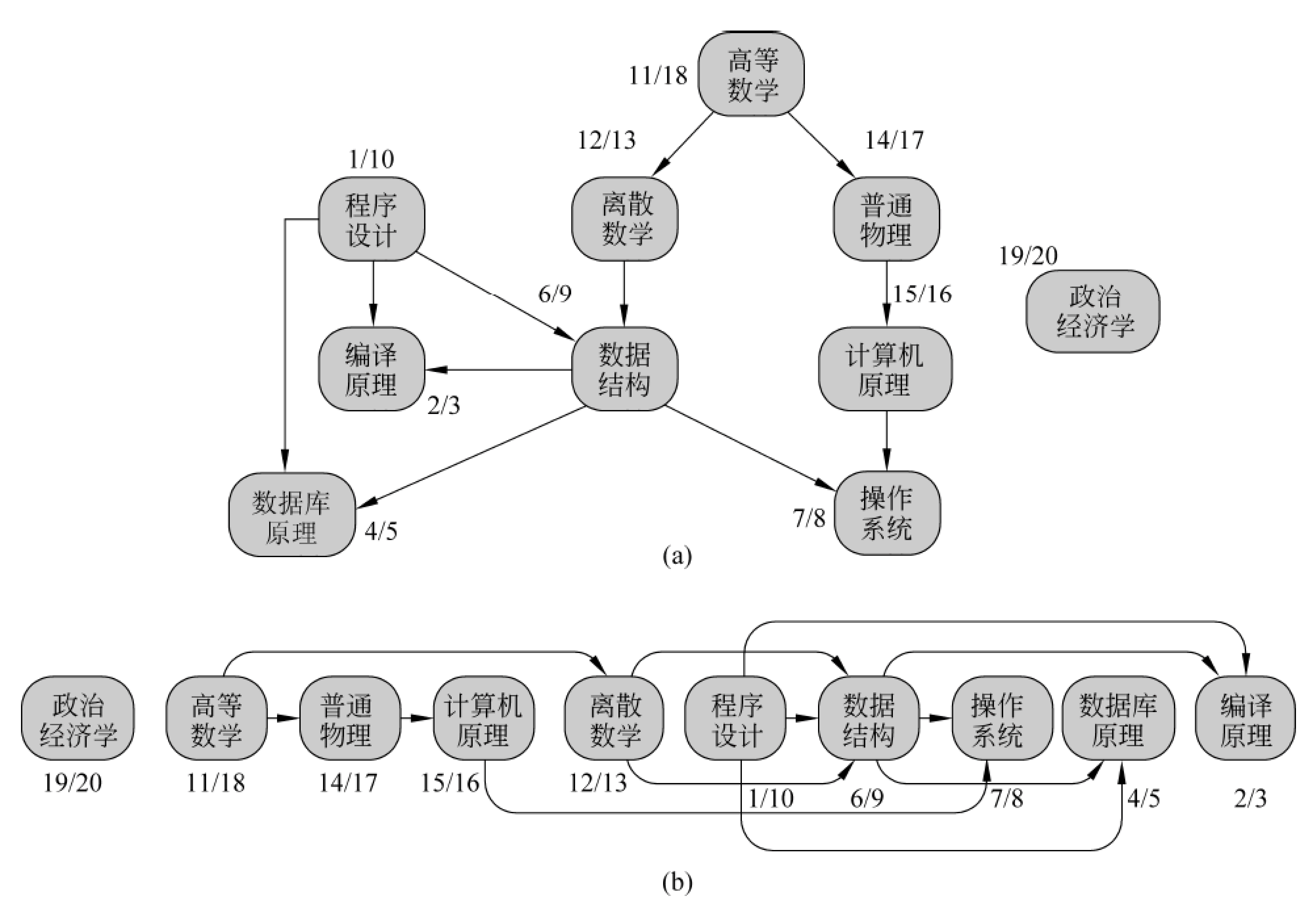


图 10-5　拓扑排序实例

2）**算法的伪代码描述**

由于在图的深度优先搜索过程中,顶点间的前辈与后代的关系就是按照边的指向引导的。顶点的完成时间 f 标示出了顶点间前辈/后代关系：若 u 是 v 的前辈,则 $f[u]>f[v]$。在 DFS 中也意味着 u 在 v 之前出栈。也就是说,对一个有向无圈图做 DFS 的过程中,只要按照顶点出栈顺序跟踪各顶点,就可得到它们的拓扑排序。

```
TOPLOGICAL-SORT(G)
for each vertex u∈V[G]
      do color[u]←WHITE
acyclicity←true
S←∅
m←|V[G]|
for each vertex s∈V[G]
      do if color[s]=WHITE
            then color[s]←GRAY
                  PUSH(S,s)
                  while S≠∅
                        do u←TOP(S)
                              if ∃v∈ Adj[u] and color[v]=GRAY
```

```
13                    then acyclicity←false
14                if ∃v∈ Adj[u] and color[v]=WHITE
15                    then color[v]←GRAY
16                         PUSH(S,v)
17                    else color[u]←BLACK
18                         top-logic[m]←u,m←m-1
19                         POP(S)
20 return acyclicity and top-logic
```

算法 10-4 计算有向无圈图的拓扑排序的算法

在第 3 行增添了一个变量 *acyclicity*,并将其初始化为 true,作为表示图是否无圈的标志。第 5 行声明了一个变量 m 初始化为顶点数,指示当前顶点加入拓扑序列的位置。

对处于 S 栈顶的顶点 u 的邻接表中搜索白色顶点(第 15 行)前,若搜索到灰色顶点(第 13 行),这意味着搜索到一条回边,也就是说,G 有一个圈。此时,将变量 *acyclicity* 置为 false(第 13 行)。

当一个顶点完成访问在第 17 行被染成黑色,就可在第 18 行将其存储于 *toplogic*[m]中,m 减 1,保证下一个出栈的顶点排在前面,而在第 18 行将其弹出栈 S。

第 20 行返回标志 *acyclicity* 及 *top-logic*。

图 10-5(b)展示了以拓扑排序的顶点按完成时间的降序排列。

由于算法 10-4 的运行时间与算法 10-3 的运行时间一致,所以,可以在时间 $\Theta(V+E)$ 内计算有向无圈图 $G=\langle V,E\rangle$ 的拓扑排序。

3. 程序实现

考虑算法中第 12 行的检测条件 $\exists v\in Adj[u]$ *and* $color[v]$=GRAY 及第 14 行的检测条件 $\exists v\in Adj[u]$ *and* $color[v]$=WHITE 的实现。我们知道 ∃ 是存在量词,它意味着"存在……"。这在程序中就意味着要在 $Adj[u]$中进行查找。最简单的办法就是线性查找法:依次检测其中的每一个元素,直至找到符合条件的元素为止。并且这两个条件实际上是在同一个集合——顶点 u 的邻接表 $Adj[u]$中考察灰色顶点(前者)或白色顶点(后者)的存在性。因此,可以用一个循环结构同时完成这两个条件的检测:依次扫描 $Adj[u]$中尚未访问过的元素,直至找到白色顶点。在找到白色顶点前遇到灰色顶点,则第 12 行的条件满足,需要执行第 13 行的操作。找到白色顶点了,第 14 行的条件为真,执行第 15 行和第 16 行的操作。

```
pair toplogicalSort(Graph *g){
    Stack *S=createStack(sizeof(int));
    int u,s,n=g->n,v,m=n,*acyclicity=(int*)malloc(sizeof(int)),
        *topLogic=(int*)malloc(n*sizeof(int));
    Color *color=(Color*)malloc(n*sizeof(Color));
    ListNode **pos=(ListNode**)malloc(n*sizeof(ListNode*));
    for(u=0;u<n;u++){
        color[u]=WHITE;
        pos[u]=g->adj[u]->nil->next;
```

```
10     }
11     *acyclicity=1;
12     for(s=0;s<n;s++){
13         if(color[s]==WHITE){
14             color[s]=GRAY;
15             push(S,&s);
16             while(!stackEmpty(S)){
17             ListNode *p;
18             u=*(int *)(S->top->key);
19             p=pos[u];
20             v=(p!=g->adj[u]->nil)? ((vertex *)(p->key))->index:-1;
21             while(p!=g->adj[u]->nil&&color[v]!=WHITE){
22                 if(color[v]==GRAY)
23                     *acyclicity=0;
24                 p=p->next;
25                 v=(p!=g->adj[u]->nil)? ((vertex *)(p->key))->index:-1;
26             }
27             pos[u]=p;
28             if(pos[u]!=g->adj[u]->nil){
29                     color[v]=GRAY;
30                     push(S,&v);
31             }else{
32                     color[u]=BLACK;
33                     topLogic[--m]=u;
34                     pop(S);
35             }
36           }
37         }
38       }
39       free(pos);free[color];
40       return make_pair(acyclicity,topLogic);
41 }
```

程序 10-3　实现算法 10-4 的 C 源代码

对程序 10-3 的说明如下。

(1) 和算法一样，函数 toplogicalSort 只有一个参数，图的邻接表 g，返回计算所得无圈有向图标志 acyclicity 和数组 topLogic 构成的 pair 型对象。

(2) 第 3 行声明的指针变量 acyclicity 对应于算法中同名的无圈有向图标志，初始化为 true。第 4 行和第 5 行声明的数组 toplogic 与 color 对应于算法中同名的顶点拓扑顺序数组和顶点颜色数组。

(3) 第 12～38 行的 **for** 循环对应算法 10-4 中的第 6～19 行，计算 *G* 的拓扑排序的操作。其中第 21～26 行的 **while** 循环行实现算法中的第 13 行搜索与顶点 u 的相邻白色顶点，此间，若遇到回边(在第 22 检测到)，则将无圈标志 acyclicity 置为 false。这样就实现了算法 10-4 中第 12 行和第 13 行的检测回边的操作。

(4) 第 33 行将完成访问的顶点 u 加入到拓扑顺序数组 topLogic 中。第 40 行将 acyclicity 和 topLogic 封装在一个 pair 对象中，然后返回。

为便于代码重用，程序 10-3 存储在文件夹 graph 中的头文件 topsort. h(函数原型声明)及源文件 topsort. c(函数定义)中。

10.1.4 应用——全排序

Sorting It All Out

An ascending sorted sequence of distinct values is one in which some form of a less-than operator is used to order the elements from smallest to largest. For example, the sorted sequence A, B, C, D implies that A＜B, B＜C and C＜D. in this problem, we will give you a set of relations of the form A＜B and ask you to determine whether a sorted order has been specified or not.

Input

Input consists of multiple problem instances. Each instance starts with a line containing two positive integers n and m, the first value indicated the number of objects to sort, where $2\leqslant n\leqslant 26$. The objects to be sorted will be the first n characters of the uppercase alphabet. The second value m indicates the number of relations of the form $A<$B which will be given in this problem instance. Next will be m lines, each containing one such relation consisting of three characters: an uppercase letter, the character "＜" and a second uppercase letter. No letter will be outside the range of the first n letters of the alphabet. Values of $n=m=0$ indicate end of input.

Output

For each problem instance, output consists of one line. This line should be one of the following three:

Sorted sequence determined after xxx relations: yyy…y.

Sorted sequence cannot be determined.

Inconsistency found after xxx relations.

where xxx is the number of relations processed at the time either a sorted sequence is determined or an inconsistency is found, whichever comes first, and yyy…y is the sorted, ascending sequence.

Sample Input

```
4 6
A<B
A<C
B<C
C<D
B<D
A<B
```

```
3 2
A<B
B<A
26 1
A<Z
0 0
```

Sample Output

Sorted sequence determined after 4 relations：ABCD.

Inconsistency found after 2 relations.

Sorted sequence cannot be determined.

1. 问题描述与分析

此问题的输入案例是 n 个对象间的 m 个“<”关系，要求由此判断这 n 个对象是否存在一个全序，也就是能否将此 n 个对象按彼此的“<”关系排成一列。如果答案是肯定的，则报告这 m 个关系中最先能确定这一先后序列的关系数。否则，若 m 个关系中如果存在矛盾的关系则报告首次产生矛盾的关系序号，如果这 m 个关系不足以判断是否所有的对象都能参与前后排列，则报告无法确定信息。例如，在第一个输入样例中，所有 6 个关系都不相互矛盾，且前 4 个关系已涉及所有 4 个对象，所以输出由前 4 个关系决定了 ABCD 的顺序。对第二个输入实例，由于两个关系是相互矛盾的，所以输出报告由前两个关系得出矛盾的信息。第三个输入实例应当有 26 个对象，但仅给出一对对象之间的关系，不足以确定所有对象的前后顺序，所以报告不确定信息。

把问题形式化为如下。

输入：一组用字母 A、B、…，表示的 n 个对象（$1\leqslant n\leqslant 26$），m 个表示这 n 个对象中的 m 对对象之间的“<”关系。

输出：若这 m 个关系不产生对象间的顺序矛盾，且前 m_1（$\leqslant m$）个关系足以说明这 n 个对象的前后顺序，则报告前 m_1 个关系确定对象的前后顺序，并输出这个顺序。若这 m 个关系存在矛盾，则报告首次出现矛盾的关系序号。若 m 个关系不足以确定这 n 个对象的先后顺序，则报告不确定信息。

如果把输入中的对象抽象为顶点，对象间的关系抽象成顶点间的有向边，则输入实例就构成一个有向图。这个问题与讨论的有向图是否为 DAG，并对 DAG 做拓扑排序的问题相关，关系间的矛盾意味着图中有圈存在。如果 m 个关系相容，意味着该有向图是一个 DAG，所以可以计算出它的拓扑排序。然而，不同的是：首先，对于有圈图，需要报告输入中的第几个关系将产生矛盾？对于 DAG，如果其中包含独立顶点，虽然拓扑排序存在，但在我们的问题中认为是不确定的情形。最后，对于有确切解的情形，还需要知道 m 中最先能确定解的 m_1 个关系。

算法思想是，将对应于关系表达式的图的边逐一插入图中并检验是否有圈。在此过程中记录下首次排序成功的对关系编号，首次失败的关系编号，或最终得出不能判断的结论（关系）。

```
SORTING-IT-ALL-OUT(R)
success←0,failed←∞,count←0
for i←1 to m
  do (u,v)←R_i 对应的边
     if failed>m
        then add (u,v) to G
           (acyclicity,top-logic)←TOPLOGICAL-SORT(G)
           if acyclicity=true
              then 将 u、v 中未曾访问过的个数累加到 count
                 if count 首次达到 n then success←i
              else failed←i
if failed≤m
  then print "Inconsistency found after " failed " relations. "
  else if count<n
          then print "Sorted sequence cannot be determined. "
          else print "Sorted sequence determined after",success
               print "relations:",top-logic
```

算法 10-5　解决 Sorting it all out 问题的过程

2. 程序实现

在 C 语言中,下列程序实现算法 10-5,解决 Sorting it all out 问题。

```
int main(){
  int n,m;
  FILE *f1,*f2;
  assert(f1=fopen("chap10/SortingItAllOut/inputdata.txt","r"));
  assert(f2=fopen("chap10/SortingItAllOut/outputdata.txt","w"));
  fscanf(f1,"%d%d",&n,&m);                     /*读取案例中对象个数 n 及关系数 m*/
  while(n || m){
    Graph *g=trivialGraph(n);                  /*创建 n 个顶点的零树*/
    char line[4];
    pair p={NULL,NULL};
    int u,v,i=0,count=0,success=0,failed=100,
        *accessed=(int*)calloc(n,sizeof(int)); /*顶点访问标志*/
    for(i=1; i<=m; i++){                       /*处理本案例的每一个关系*/
      fscanf(f1,"%s",line);                    /*读取关系*/
      if(failed<=m)                            /*已经知道失败*/
        continue;
      u=line[0]-'A';v=line[2]-'A';             /*计算关系中对象对应的顶点*/
      addEdg(g,u,v,1.0);                       /*向图中添加边*/
      if(p.first)free(p.first);
      if(p.second)free(p.second);
      p=toplogicalSort(g);                     /*计算图的拓扑排序*/
      if(*((int*)p.first)==0){                 /*若是有圈图,排序失败*/
        failed=i;
```

```
24             continue;
25          }
26          if(accessed[u]==0){                       /* 计数访问过的顶点(对象) */
27             accessed[u]=1;
28             count++;
29          }
30          if(accessed[v]==0){                       /* 计数访问过的顶点(对象) */
31             accessed[v]=1;
32             count++;
34          }
35          if(count<n)                               /* 计数成功步骤 */
36             success++;
37       }
38       if(failed<=m)                                /* 排序失败 */
39          fprintf(f2,"Inconsistency found aftter %d relations.\n",failed);
40       else if(count<n){                            /* 不能排序 */
41          fprintf(f2,"Sorted sequence cannot be determined.\n");
42       }else{                                       /* 排序成功 */
43          int k, *s=(int *)p.second;
44          fprintf(f2,"Sorted sequence determined after %d relations:",success+1);
45          for(k=0;k<n;k++){
46             fputc('A'+s[k],f2);
47          }
48          fputc('\n',f2);
49       }
50       free(accessed);free(p.first);free(p.second);
51       fscanf(f1,"%d%d",&n,&m);
52    }
53    fclose(f1);fclose(f2);
54    return 0;
55 }
```

程序 10-4　解决 Sorting it all out 问题的 C 程序

对程序 10-4 的说明如下。

(1) 由于输入文件含有若干个案例数据，第 7～52 行的 **while** 循环依次处理每个案例。

(2) 对每一个案例，第 8 行调用函数 zeroGraph 创建一个零图(只有顶点没有边的图)。第 12 行声明一个用来标识顶点是否已访问过的数组 accessed，其中的元素初始化为 0(注意分配存储空间时调用的函数是 calloc)。

(3) 第 13～37 行的 **for** 循环对应算法过程中第 2～10 行的操作，逐条处理输入案例中的每一个关系。第 14 行读取关系表达式，在以前处理的关系均合法的前提下(第 15 行和第 16 行所做检测)，第 17 行将关系表达式中的两个对象分别转换成对应的图中顶点 u 和 v。第18 行调用函数 addEdge 将边(u,v)加入到图 g 中。第 21 行调用函数 toplogicalSort 计算图 g 是否无圈。对有圈的情形，第 22～25 行负责记载失败步骤 failed。第 26～34 行根据数组 accessed 跟踪的信息计算 u、v 中未曾访问过的顶点数，累计于 count 中。第 38～49 行实

现算法中第 11～16 行的操作,分情况向输出文件写入相关信息。

程序存储在文件夹 chap10/Sorting It All Out 中的源文件 sortingitallout.c 中,读者可打开文件研读,并试运行。

10.2 有向图的强连通分支

10.2.1 算法描述与分析

1. 问题的理解与描述

现在来考虑深度优先搜索的另一个经典应用:将一个有向图分解为强连通分支。本节说明如何利用两次深度优先搜索来做到这一点。很多对有向图的算法都以这样的分解作为开头。分解后,算法分别对每一个强连通分支运行。问题的解按各分支间的连接结构合并而成。

有向图 $G=(V,E)$的一个**强连通分支** C 是一个使得其中每一对顶点 u 和 v 均有 u v 及 v u,即顶点 u 和 v 是相互可达的顶点集合 $C\subseteq V$。图 10-6 展示了一个例子。

图 10-6(a)为一个有向图 G。G 的强连通分支显示为阴影区域。每一个顶点标示出了它的发现时间和完成时间。树枝边也加有阴影。图 10-6(b)为图 G^T(G 的转置)。展示了 STRONGLY-CONNECTED-COMPONENTS 的第 3 行计算的深度优先森林,树枝边也加有阴影。每一个强连通分支对应一棵深度优先树。加了深色阴影的顶点 b、c、g 和 h 是由对 G^T 进行深度优先搜索而产生的深度优先树的根。图 10-6(c)将 G 中的各强连通分支收缩为单一顶点后得到的分支图。图 10-6(d)为图 G^T 的分支图。

计算强连通分支的问题形式化表示为如下。

输入:有向图 G。

输出:G 的各强连通分支$\{C_1,C_2,\cdots,C_k\}$。

寻求图 $G=<V,E>$的强连通分支的算法要用到 G 的转置,它定义于本章的开头。也就是图 $G^T=<V,E^T>$,其中 $E^T=\{(u,v):(v,u)\in E\}$,即 E^T 是由 G 的所有边的反向而得。给定 G 的一个邻接表表示,创建 G^T 的时间是 $\Theta(V+E)$(见算法 10-1 和算法 10-2)。有趣的是 G 和 G^T 恰有相同的连通分支:u 和 v 在 G 中相互可达当且仅当它们在 G^T 中相互可达。图 10-6(b)展示了图 10-6(a)中的图的转置,其中的强连通分支带有阴影。如果把一个有向图 G 中的各个强连通分支收缩为一个“顶点”,则将得到一个称为 G 的**分支图**的有向图。图 10-6(c)展示了图 10-6(a)的分支图。很容易理解引理 10-1。

引理 10-1 有向图的分支图是一个有向无圈图。

这是因为如果分支图中存在一个圈,则该圈又构成 G 的一个更大的连通分支,此与分支图的定义不符。

假定对有向图 G 做了一次 DFS,则每个顶点 u 都具有发现时间 $d[u]$和完成时间 $f[u]$。设 G 有强连通分支 $C_1,C_2,\cdots,C_k$。对 G 的分支图中的各个顶点(G 的各个强连通分支 C_i,

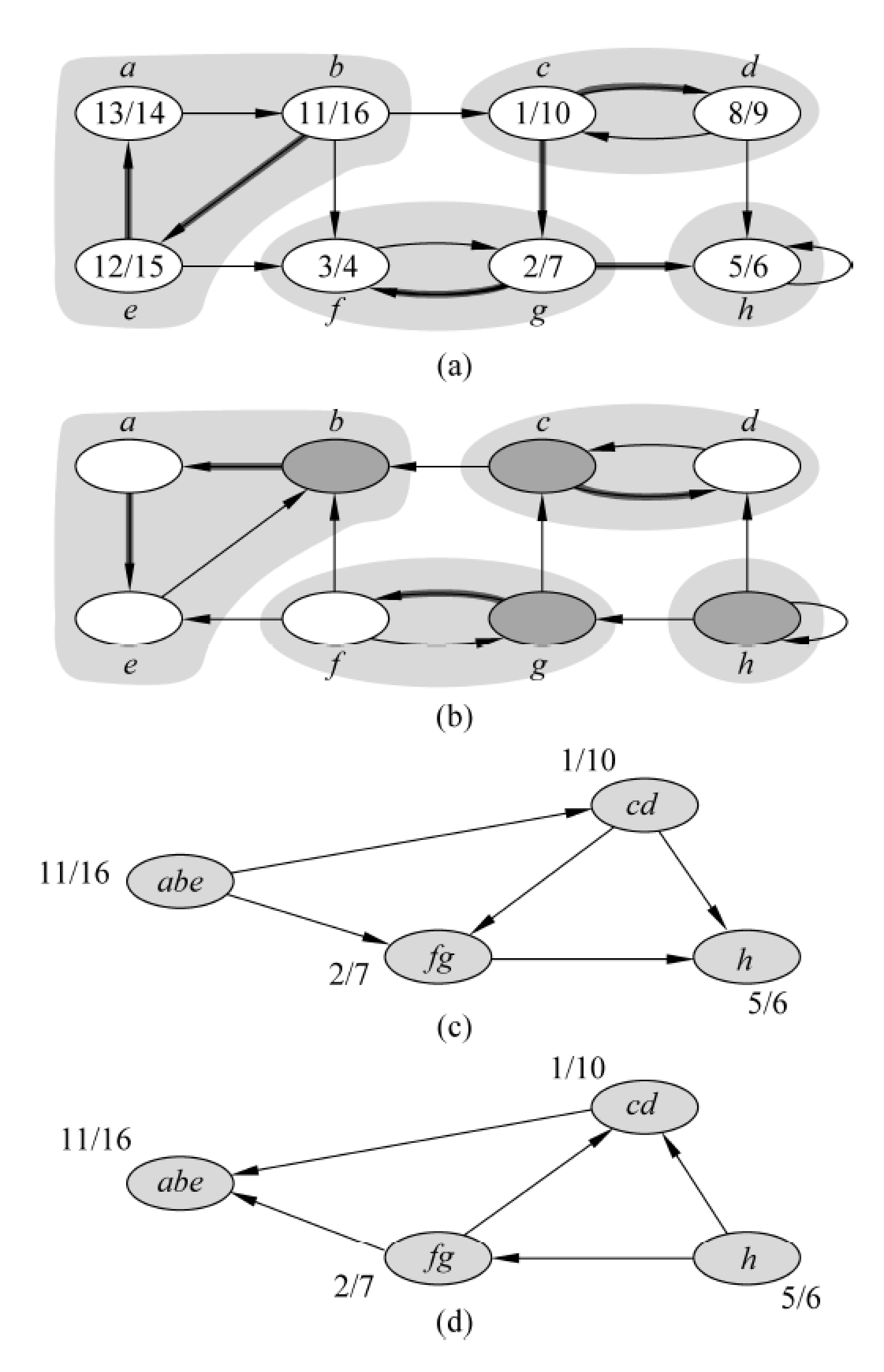


图 10-6　有向图 G 及其转置的分支图

$i=1,2,\cdots,k$)，定义发现时间和完成时间 $d(C_i)=\min_{u\in C_i}\{d[u]\}$ 及 $f(C_i)=\max_{u\in C_i}\{f[u]\}$。例如，图 10-6(c)中各个顶点所标示的发现时间、完成时间。有趣的是，分支图中边的方向都是从完成时间较大的顶点指向完成时间较小的顶点。这不是偶然现象，由引理 6-4 可知，有向图的分支图是无环的，所以图中顶点按完成时间的降序恰为顶点的拓扑顺序。由此可以得到引理 10-2。

引理 10-2　设 C 和 C' 是有向图 $G=(V,E)$ 的两个不同的强连通分支。在一次深度优先搜索中其完成时间分别为 $f(C)$ 和 $f(C')$。若有边 $(u,v)\in E^T$，其中 $u\in C$ 及 $v\in C'$，则 $f(C)<f(C')$。

例如，图 10-6(d)可视为图 10-6(c)的转置，其实也是图 10-6(a)的转置的分支图。其中所有顶点旁边标示的是图 10-6(c)的顶点的发现时间/完成时间，而所有的边都是从完成时间的较大者指向完成时间的较小者。

2. 算法的伪代码描述

利用引理 10-2，用下列的线性时间(即 $\Theta(V+E)$ 时间)算法计算有向图 $G=(V,E)$ 的强连通分支，它利用两次深度优先搜索，一次对 G，一次对 G^T。

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  调用 DFS(G)对每个顶点 u 计算 f[u]
2  计算 G^T
3  调用 DFS(G^T),但在 DFS 的主循环中按(以在第 1 行中的计算)f[u]的降序进行
4  输出第 3 步所得的深度优先森林中的每一棵树中的顶点作为一个强连通分支
```

算法 10-6　计算有向图的强连通分支的算法

第 3 步按在第 1 步中所得的顶点的完成时间降序作为第二次 DFS 的主循环顺序,在进入一个连通分支进行深度优先搜索,将完成该分支中的所有顶点的访问,形成一棵深度优先树。根据引理 10-2,此次搜索不会进入另一个强连通分支,因为完成时间较大的分支在分支图中没有指向完成时间较小的分支。所以完成一个分支的搜索后,主循环会进入下一轮重复,进行另一个分支的搜索,形成又一棵深度优先树,……。第 4 步将对应强连通分支的搜索树中顶点输出刚好得到每个分支。

从上述说明中可见,STRONGLY-CONNECTED-COMPONENTS 过程中调用的 DFS 需要做一点修改:使它能够按照一定的顺序执行主循环。

```
DFS-BY-ORDER(G,order)
1  for each vertex u∈V[G]
2        do color[u]←WHITE
3           π[u]←NIL
4  S←∅
5  m←|V[G]|
6  for s←1 to V[G]
7      do if color[order[s]]=WHITE
8         then color[order[s]]←GRAY
9              PUSH(S,order[s])
10             while S≠∅
11                 do u←TOP(S)
12                    if ∃v∈ Adj[u] and color[v]=WHITE
13                       then color[v]←GRAY
14                            π[v]←u
15                            PUSH(S,v)
16                       else color[u]←BLACK
17                            top-logic[m]←u,m←m-1
18                            POP(S)
19     return π and top-logic
```

算法 10-7　按指定顺序进行主循环的 DFS 算法

与 DFS 相比,DFS-BY-ORDER 多了一个指示访问顶点顺序的数组 *order* 作为参数,其中的元素是 1,2,…,n 的一个排列。第 6～19 行的主循环 **for** 的重复顺序为 *order*[1],*order*[2],…,*order*[n]。

介于第 9 行和第 17 行之间的代码与算法 10-4 的基本一致,只是删除对无圈图标志 *acyclicity* 的操作。第 18 行将计算所得的数组 π、*top-logic* 返回。

3. 强连通分支的输出

为了能根据对 G^T 运行 DFS-BY-ORDER 返回的数组 π 输出 G 的各个强连通分支，给出如下代码。

```
MAKE-FOREST(π)
1  n←length[π]
2  forest←∅
3  for s←1 to n
4      do if π[s]=NIL
5          then tree←{s}
6               for v←1 to n
7                   do if IS-ROOT(π,s,v)
8                          then add v to tree
9               add tree to forest
10 return forest
```

算法 10-8　根据数组 π 创建深度优先森林的过程

过程 MAKE-FOREST 运行如下。第 3～9 行的 **for** 循环扫描数组 π 中的树根结点 s，一个根结点，确定森林中一棵树。内嵌的第 6～8 行的 **for** 循环对每个顶点 v 检测是否在以 s 为根的树中，若是，则将 v 加入树中。第 9 行将以 s 为根的树加入到森林中。可以用嵌套的链表来组织 MAKE-FOREST 返回的数据 *forest*（见图 10-7）。

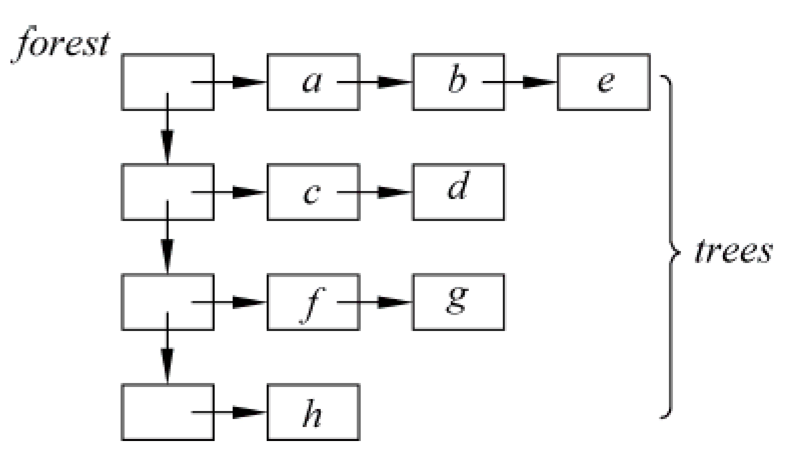
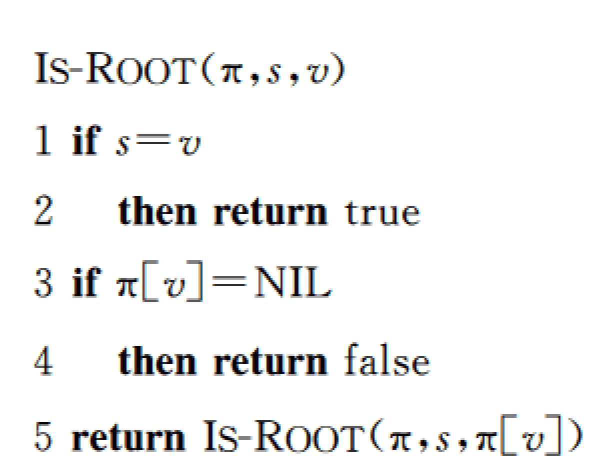


图 10-7　将 MAKE-FOREST 返回的数据组织成嵌套的链表

过程中第 7 行用来测试顶点 s 是否为 v 的根的 IS-ROOT 过程的伪代码如下。



```
IS-ROOT(π,s,v)
if s=v
    then return true
if π[v]=NIL
    then return false
return IS-ROOT(π,s,π[v])
```

算法 10-9　检测顶点 s 是否为顶点 v 的根的过程

本过程用递归的方式判断 s 到 v 是否构成搜索树中的一条路径。由于 IS-ROOT 至多递归 n 次，所以它的时间复杂度为 $\Theta(n)$。于是，过程 PRINT-COMPONENTS 的时间复杂度为 $m \cdot \Theta(n^2)$。其中，m 是森林中的搜索树数。

利用 DFS-BY-ORDER、TRANSPOSE-DIRECTED-GRAPH 和 MAKE-FOREST，将 STRONGLY-CONNECTED-COMPONENTS 重写如下。

```
STRONGLY-CONNECTED-COMPONENTS(G)
for u←1 to n
    do order[u]←u
```

```
order←调用 DFS-BY-ORDER (G,order)返回的 top-logic 数组
G^T←TRANSPOSE-DIRECTED-GRAPH (G)
π←调用 DFS-BY-ORDER (G^T,order)返回的 π 数组
return MAKE-FOREST (π)
```

算法 10-10 重写过程 STRONGLY-CONNECTED-COMPONENTS

10.2.2 程序实现

由于算法是基于 DFS 的,所以对程序 10-2 中定义的函数 dfs 稍做修改就可以实现能按照指定顺序执行主循环的 DFS-BY-ORDER 算法。

```
pair dfsByOrder(Graph *g,int *order){
    Stack *S=createStack(sizeof(int));
    int u,s,n=g->n,v,m=n,
        *pi=(int*)malloc(n*sizeof(int)),
        *topLogic=(int*)malloc(n*sizeof(int));
    Color *color=(Color*)malloc(n*sizeof(Color));
    ListNode **pos=(ListNode**)malloc(n*sizeof(ListNode*));
    for(u=0;u<n;u++){
        pi[u]=-1;
        color[u]=WHITE;
        pos[u]=g->adj[u]->nil->next;
    }
    for(s=0;s<n;s++){
        if(color[order[s]]==WHITE){
            color[order[s]]=GRAY;
            push(S,&order[s]);
            while(!stackEmpty(S)){
                ListNode *p;
                u=*(int*)(S->top->key);
                p=pos[u];
                v=(p!=g->adj[u]->nil)?((vertex*)(p->key))->index:-1;
                while(p!=g->adj[u]->nil&&color[v]!=WHITE){
                    p=p->next;
                    v=(p!=g->adj[u]->nil)?((vertex*)(p->key))->index:-1;
                }
                pos[u]=p;
                if(pos[u]!=g->adj[u]->nil){
                    color[v]=GRAY;
                    pi[v]=u;
                    push(S,&v);
                }else{
                    color[u]=BLACK;
                    topLogic[--m]=u;
```

```
                    pop(S);
                }
            }
        }
    }
    free(pos);free(color);
    return make_pair(pi,topLogic);
}
```

程序 10-5　实现按指定主循环重复顺序的 DFS 算法 10-7 的 C 源代码

程序 10-5 的代码与程序 10-3 十分相近，删除无圈图标志的操作，保留数组 pi，并作为返回数据之一。读者可比较研读，在此不再赘述。

在计算有向图强连通分支的算法中需要计算图 G 的转置 G^T，这个过程定义于本章开头的算法 10-2 中，在 C 语言中把它实现为一个函数。

```
1 Graph *transpose(Graph *g){
2     Graph *gt=(Graph*)malloc(sizeof(Graph));
3     int u,v,n=gt->n=g->n;
4     gt->adj=(LinkedList**)malloc(n*sizeof(LinkedList));
5     for(u=0;u<n;u++)
6         gt->adj[u]=createList(sizeof(vertex),NULL);
7     for(u=0;u<n;u++){
8         ListNode *i=g->adj[u]->nil->next;               /*g 的顶点 u 的邻接表指针*/
9         while(i!=g->adj[u]->nil){                      /*在 u 的邻接表扫描*/
10            vertex *u1=(vertex*)malloc(sizeof(vertex));
11            v=((vertex *)(i->key))->index;               /*u、v 邻接*/
12            u1->index=u;
13            u1->weight=((vertex *)(i->key))->weight;
14            listPushFront(gt->adj[v],u1);
15            i=i->next;
16        }
17    }
18    return gt;
19 }
```

程序 10-6　实现计算图的转置的算法 10-2 的 C 源代码

对程序 10-6 的说明如下。

(1) 函数 transpose 只有一个参数：图 G 的邻接表表示 g。它将返回 G 的转置 G^T 的邻接表表示 gt。

(2) 第 2～6 行为图 g 的转置图 gt 分配空间。其顶点数与图 g 的顶点数相同。

(3) 第 7～17 行实现的是算法 10-2 中的第 3～5 行的操作：根据 G 中的边 (u,v) 在 G^T 中插入边 (v,u)。具体地说，就是找到 G 中顶点 u 的邻接表中的顶点 v，然后在 G^T 的顶点 v 的邻接表中加入顶点 u。

为便于代码重用，把程序 10-6 添加到文件夹 graph 中的头文件 graph.h(原型声明)及

源文件 graph.c(定义代码)文件中。

做好这些准备后,就可以把 STRONGLY-CONNECTED-COMPONENTS 过程实现为下列函数。

```
int isRoot(int * pi,int s,int v){
    if(s==v)                                      /* 树根本身 */
        return 1;
    if(pi[v]==-1)                                 /* 另一棵树的根 */
        return 0;
    return isRoot(pi,s,pi[v]);                    /* 递归检测父结点 */
}
LinkedList * makeForest(int * pi,int n){
    LinkedList * forest=createList(sizeof(LinkedList * ),NULL);
    int s,v;
    for(s=0;s<n;s++){
        if(pi[s]==-1){                            /* 是一棵树的根 */
            LinkedList * tree=createList(sizeof(int),NULL);
            for(v=0;v<n;v++)                      /* 存储树中的结点 */
                if(isRoot(pi,s,v))
                    listPushBack(tree,&v);
            listPushBack(forest,&tree);
        }
    }
    return forest;
}
LinkedList * strongConnectedComponents(Graph * g){
    int n=g->n,u;
    pair r;
    int * pi, * order=(int * )malloc(n * sizeof(int));
    LinkedList * forest;
    Graph * gt;
    for(u=0;u<n;u++)
        order[u]=u;
    gt=transpose(g);
    r=dfsByOrder(g,order);
    free(r.first);free(order);
    order=(int * )(r.second);
    r=dfsByOrder(gt,order);
    graphClear(gt);free(gt);
    pi=(int * )(r.first);free(r.second);free(order);
    forest=makeForest(pi,n);
    free(pi);
    return forest;
}
```

程序 10-7　实现算法 10-8～算法 10-10 的 C 源代码

对程序 10-7 的说明如下。

(1) 第 1～7 行定义的函数 isRoot 实现的是算法 10-9 IS-ROOT。代码与算法的伪代码几乎一致，在此不再赘述。

(2) 第 8～21 行定义的函数 makeForest 实现的是算法 10-8 MAKE-FOREST。除了参数多了一个说明图的顶点数的 n 以外，实现代码也是十分直接的。要注意的是，第 9 行将 forest 声明为 LinkedList 型指针，而第 13 行将 tree 也声明成 LinkedList 型指针。当第 14～16 行将以 s 为根的深度优先树中的所有结点加入到 tree 中后，第 17 行将 tree 插入到 forest 中。

(3) 第 22～40 行定义的函数 strongConnectedComponents 实现的是算法 10-10 STRONGLY-CONNECTED-COMPONENTS。第 31 行调用 dfsByOrder 函数按在第 28 行和第 29 行初始化为自然顺序的 order 对 g 做 DFS。第 33 行将封装在返回结果 r 中的拓扑顺序数组 topLogic 赋予 order。第 34 行对 gt 调用 dfsByOrder 按 g 的拓扑顺序 order 做 DFS。第36 行将封装在返回结果 r 中的父结点指针数组赋予数组 pi，第 37 行调用 makeForest 计算出由强连通分支组成的深度优先森林 forest，第 39 行将其返回。

为便于代码重用，将程序 10-6 添加到文件 graph. h 和 graph. c 中，而将程序 10-5、程序 10-7 存储在文件夹 graph 中的文件 scc. h 和 scc. c 中。

10.2.3　应用——亲情号

Quote Calling Circles

Description

If you've seen television commercials for long-distance phone companies lately, you've noticed that many companies have been spending a lot of money trying to convince people that they provide the best service at the lowest cost. One company has "quotcalling circles." You provide a list of people that you call most frequently. If you call someone in your calling circle (who is also a customer of the same company), you get bigger discounts than if you call outside your circle. Another company points out that you only get the big discounts for people in your calling circle, and if you change who you call most frequently, it's up to you to add them to your calling circle.

LibertyBell Phone Co. is a new company that thinks they have the calling plan that can put other companies out of business. LibertyBell has calling circles, but they figure out your calling circle for you. This is how it works. LibertyBell keeps track of all phone calls. In addition to yourself, your calling circle consists of all people whom you call and who call you, either directly or indirectly. For example, if Ben calls Alexander, Alexander calls Dolly, and Dolly calls Ben, they are all within the same circle. If Dolly also calls Benedict and Benedict calls Dolly, then Benedict is in the same calling circle as Dolly, Ben, and Alexander. Finally, if Alexander calls Aaron but Aaron doesn't call Alexander, Ben, Dolly, or Benedict, then Aaron is not in the circle.

You've been hired by LibertyBell to write the program to determine calling circles given a log of phone calls between people.

Input

The input file will contain one or more data sets. Each data set begins with a line containing two integers, n and m. The first integer, n, represents the number of different people who are in the data set. The maximum value for n is 25. The remainder of the data set consists of m lines, each representing a phone call. Each call is represented by two names, separated by a single space. Names are first names only (unique within a data set), are case sensitive, and consist of only alphabetic characters; no name is longer than 25 letters. For example, if Ben called Dolly, it would be represented in the data file as

Ben Dolly

Input is terminated by values of zero (0) for n and m.

Output

For each input set, print a header line with the data set number, followed by a line for each calling circle in that data set. Each calling circle line contains the names of all the people in any order within the circle, separated by comma-space (a comma followed by a space). Output sets are separated by blank lines.

Sample Input

```
5 6
Ben Alexander
Alexander Dolly
Dolly Ben
Dolly Benedict
Benedict Dolly
Alexander Aaron
14 34
John Aaron
Aaron Benedict
Betsy John
Betsy Ringo
Ringo Dolly
Benedict Paul
John Betsy
John Aaron
Benedict George
Dolly Ringo
Paul Martha
George Ben
Alexander George
Betsy Ringo
Alexander Stephen
Martha Stephen
```

```
Benedict Alexander
Stephen Paul
Betsy Ringo
Quincy Martha
Ben Patrick
Betsy Ringo
Patrick Stephen
Paul Alexander
Patrick Ben
Stephen Quincy
Ringo Betsy
Betsy Benedict
Betsy Benedict
Betsy Benedict
Betsy Benedict
Betsy Benedict
Betsy Benedict
Quincy Martha
0 0
```

Sample Output

```
Calling circles for data set 1:
Ben,Alexander,Dolly,Benedict
Aaron

Calling circles for data set 2:
John,Betsy,Ringo,Dolly
Aaron
Benedict
Paul,George,Martha,Ben,Alexander,Stephen,Quincy,Patrick
```

1. 问题描述与分析

本题是要求根据通信客户间相互通信(包括直接的和间接的双向通信)的关系来确定这样的客户群。例如,若 Ben 打电话给 Alexander,Alexander 打电话给 Dolly,Dolly 打电话给 Ben,那么他们就属于一个客户群。如果 Dolly 还打电话给 Benedict 且 Benedict 打电话给 Dolly,则 Benedict 也和 Dolly、Ben 及 Alexander 一样属于该客户群。然而,若 Alexander 打电话给 Aaron,但 Aaron 并不给 Alexander、Ben、Dolly 及 Benedict,则 Aaron 并不属于该客户群。移动电话公司将对这样的客户群施以资费优惠。如果把通信客户抽象为一个点,客户间的呼叫关系抽象为有向边,则一个输入实例就构成一个有向图。题目实际上是要求计算图的强连通分支,对每个分支内的用户给予优惠。所以,调用 STRONGLY-CONNECTED-COMPONENTS 就能解决 Quote Calling Circle 问题。

2. 程序实现

```
int find(char **name,char * s,int index){
```

```
    int i;
    for(i=0;i<index;i++)
        if(strcmp(name[i],s)==0)
            return i;
    return -1;
}
int main(){
    int n,m,count=0;
    FILE *f1,*f2;
    assert(f1=fopen("chap10/Quote calling circles/inputdata.txt","r"));
    assert(f2=fopen("chap10/Quote calling circles/outputdata.txt","w"));
    fscanf(f1,"%d%d",&n,&m);
    while(n || m){
        int i,index=0;
        Graph *g=zeroGraph(n);
        LinkedList *forest;
        ListNode *p;
        char **name=(char**)malloc(n*sizeof(char*));
        for(assert(name),i=0;i<n;i++)
            name[i]=(char*)calloc(26,sizeof(char));
        for(i=0;i<m;i++){
            char A[26],B[26];
            int u,v,k;
            fscanf(f1,"%s%s",A,B);
            k=find(name,A,index);
            if(k==-1){
                u=index;
                strcpy(name[index++],A);
            }else{
                u=k;
            }
            k=find(name,B,index);
            if(k==-1){
                v=index;
                strcpy(name[index++],B);
            }else{
                v=k;
            }
            addEdge(g,u,v,1.0);
        }
        forest=strongConnectedComponents(g);
        fprintf(f2,"Calling circles for data set %d:\n",++count);
        p=forest->nil->next;
        while(p!=forest->nil){
```

```
46          LinkedList * tree= * ((LinkedList**)p->key);
47          ListNode * q=tree->nil->next;
48          while(q!=tree->nil){
49             int u= * ((int * )q->key);
50             fprintf(f2,"%s ",name[u]);
51             q=q->next;
52          }
53          fputc('\n',f2);
54          clrList(tree,NULL);free(tree);
55          p=p->next;
56       }
57       clrList(forest,NULL);free(forest);
58       fprintf(f2,"\n");
59       for(i=0;i<n;i++)
60          free(name[i]);
61       free(name);
62       fscanf(f1,"%d%d",&n,&m);
63    }
64    fclose(f1);fclose(f2);
65    return 0;
66 }
```

程序 10-8　解决 Quote Calling Circle 问题的 C 程序

对程 10-8 的说明如下。

(1) 第 14～63 行的 **while** 循环处理输入文件中的每一个案例。其中，第 16 行声明了一个零图 g。第 17 行声明了一个链表 forest，用来存储 g 的强连通分支。第 19～21 行声明了一个字符串数组 name，用来存放用户的名字。

(2) 第 22～41 行的 **for** 循环逐一读取输入文件中案例数据的每一条通话信息，并将其转换成有向图。其中，第 25 行将两个通话用户名字读取到串 A 和 B，第 26～32 行检验A 是否曾经出现过(第 26 行调用函数 find 在数组 name 中查找，该函数定义在第 1～7 行)，若未曾出现过，则将其存放在 name 的 index 位置上(index 声明于第 15 行，初始化为 0，表示新的用户名字插入到 name 中的位置)，对应的图中的顶点 u 为 index。否则，u 为 A 在 name 中的位置。第 33～39 行用同样的方式处理用户名 B，并确定对应的图中顶点 v。第 40 行将边(u，v)插入到图 g 中。第 42 行调用函数 strongConnectedComponents 计算 g 的强连通分支，结果返回给 forest。

(3) 第 45～56 行的 **while** 循环向输出文件写入 forest 中的每一棵树(对应 g 的一个强连通分支)。其中，第 48～52 行的内嵌 **while** 循环将当前树中的每个结点对应的用户名字写入输出文件。

程序 10-8 存储在文件夹 chap10/Quote Calling Circle 中的源文件 QuoteCallingCircle. c 中，读者可打开研读，并试运行。

10.3 无向图的双连通分支

10.3.1 算法描述与分析

1. 问题的理解与描述

设 $G=(V,E)$是一个连通无向图。G 的一个**关节点**是移除该点将导致该图不连通的顶点。G 的一座**桥**(Bridge)是 G 的一条边,移除该边将导致 G 不连通。G 的一个**双连通分支**是一个边的最大集合,其中的任意两条边都同在一条简单环路上。图 10-8 给出了这些定义的示例。可以利用深度优先搜索来确定关节点、桥和双连通分支。设 $G_\pi=(V,E_\pi)$是 G 的一棵深度优先树。

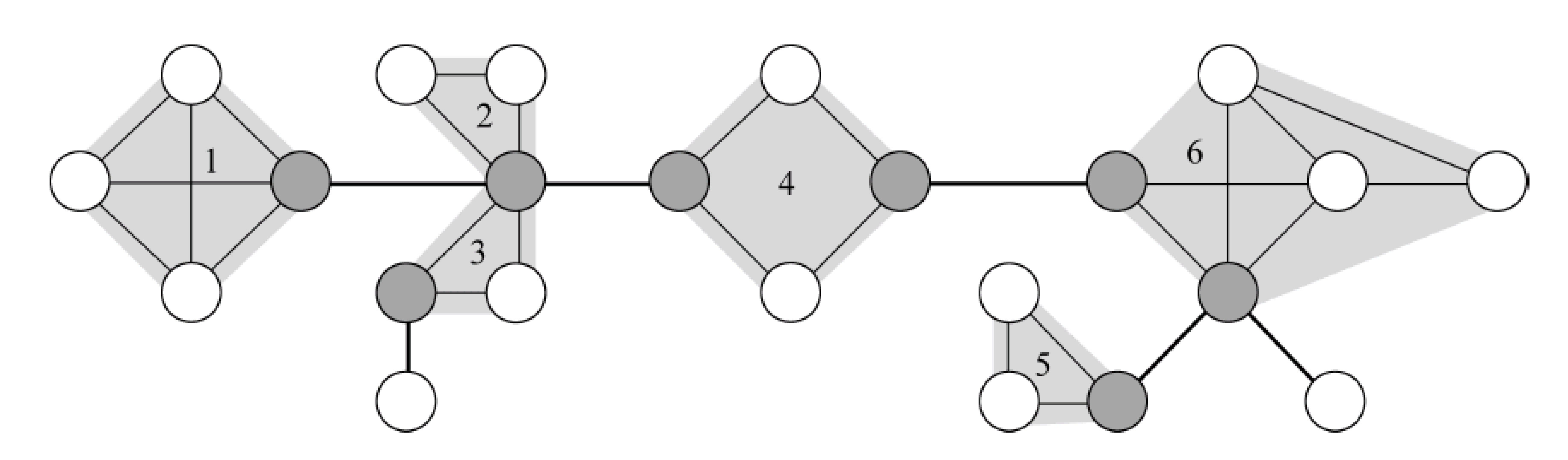


图 10-8 连通无向图的关节点、桥和双连通分支

图 10-8 所示为连通无向图的关节点、桥和双连通分支。关节点加了深色阴影,桥也加上了深色阴影,双连通分支是阴影区域中的边,并加以编号。

显然,没有关节点的图 G 是双连通图。若两个关节点 u、v 之间存在 G 的边(u,v),则(u,v)就是 G 的一座桥。删除所有的桥,得到的 G 的子图构成 G 的双连通分支。所以,计算图的关节点是一个关键问题,把它形式化为如下。

输入:无向连通图 G,G 中一个顶点 s 作为源顶点。

输出:若 G 有关节点,返回 G 的关节点构成的集合 A,否则返回空集。

2. 关节点在 DFS 过程中的性质

下面用一个简单的例子来说明关节点在 DFS 过程中的特性,并由此来设计一个利用对无向连通图的 DFS 查找关节点的算法。在图 10-9 中,图 10-9(a)所表示的无向连通图具有 c、b、g 和 h 4 个关节点(标示为灰色)。图 10-9(b)和图 10-9(c)都是对图 10-9(a)中的图进行 DFS 后形成的搜索树,结点以 d/f 标示发现时间和完成时间。图 10-9(b)的源顶点是 b,图 10-9(c)的源顶点是 a。在这两棵深度优先树中,我们观察到:

(1) 如果树根是图中的一个关节点,则它有多于一个孩子(图 10-9(b)中的情形)。

(2) 如果树中的一个非根结点 v 是图中的一个关节点,则该结点必不存在一个孩子结点 w,它有一条指向 v 的父亲的回边。

为了使 DFS 过程能跟踪顶点是否具有关节点的性质,定义深度优先树中结点 v 的

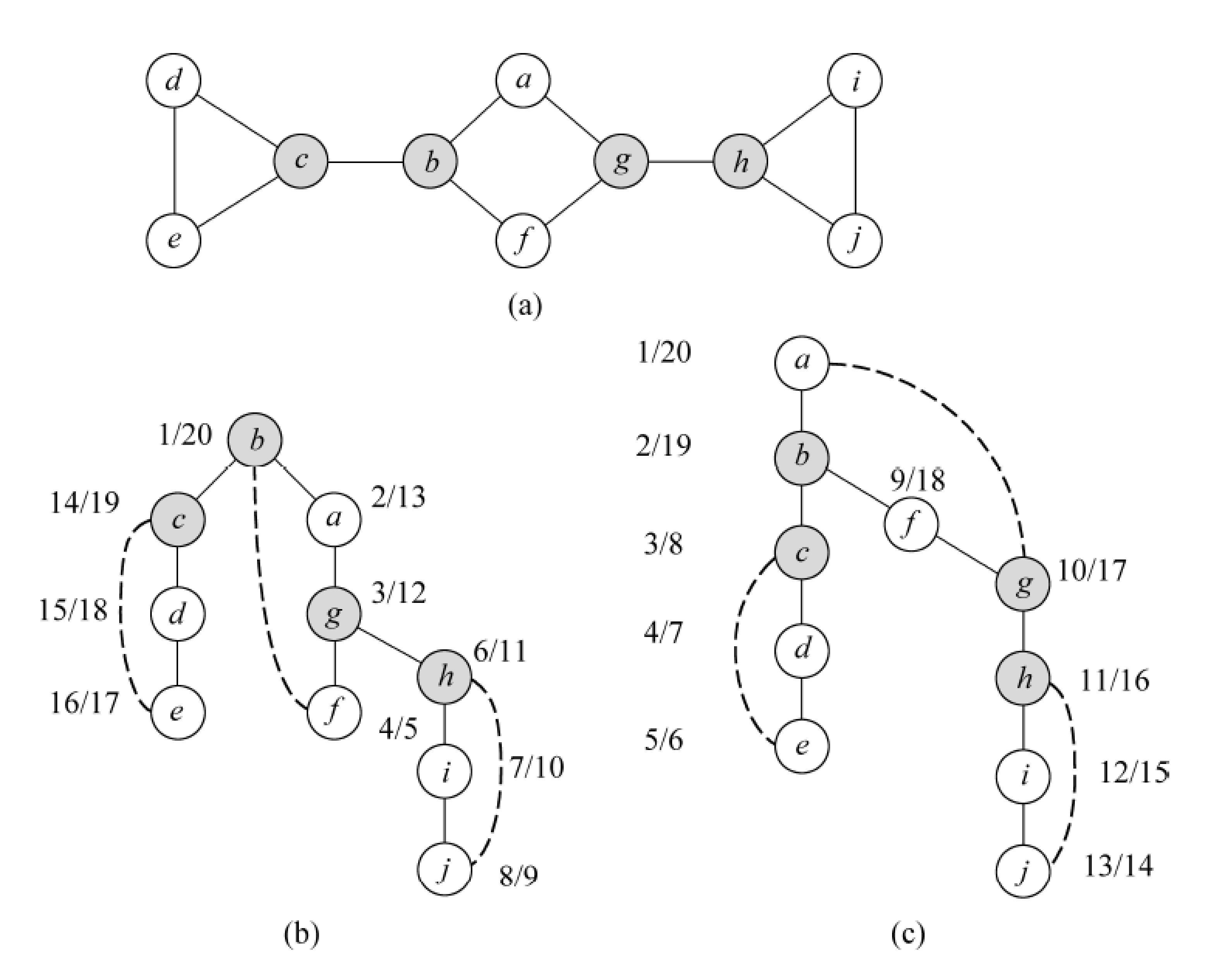


图 10-9　关节点在深度优先树中的特性

属性：

$$low[v] = \min \begin{cases} d[v] & \\ d[u] & (v,u)\ \text{为一条回边} \\ low[w] & w\ \text{是}\ v\ \text{的孩子} \end{cases} \tag{10-2}$$

对一个非根结点 v 而言，如果有它的孩子 w 的属性 $low[w]$ 不小于它的发现时间 $d[v]$，则意味着 v 不存在后代有指向 v 的前辈的回边。这样，就可判断 v 就是图中的一个关节点。按结点的 low 属性定义，将图 10-9(c)中的深度优先树中各结点的 low 属性值添加在发现时间/完成时间标示之后，这样每个结点就有标示：d/f、low，如图 10-10 所示。注意，树中结点 b 右孩子 c 的 low 属性值 3 大于 b 的发现时间 2，b 恰为图中一个关节点。同样，关节点 c 有孩子结点 d 的 low 属性不小于 c 的发现时间。此外，关节点 g 和 h 也是如此。建议读者对图 10-9(b)的深度优先树中的各个结点标示出 low 属性值。

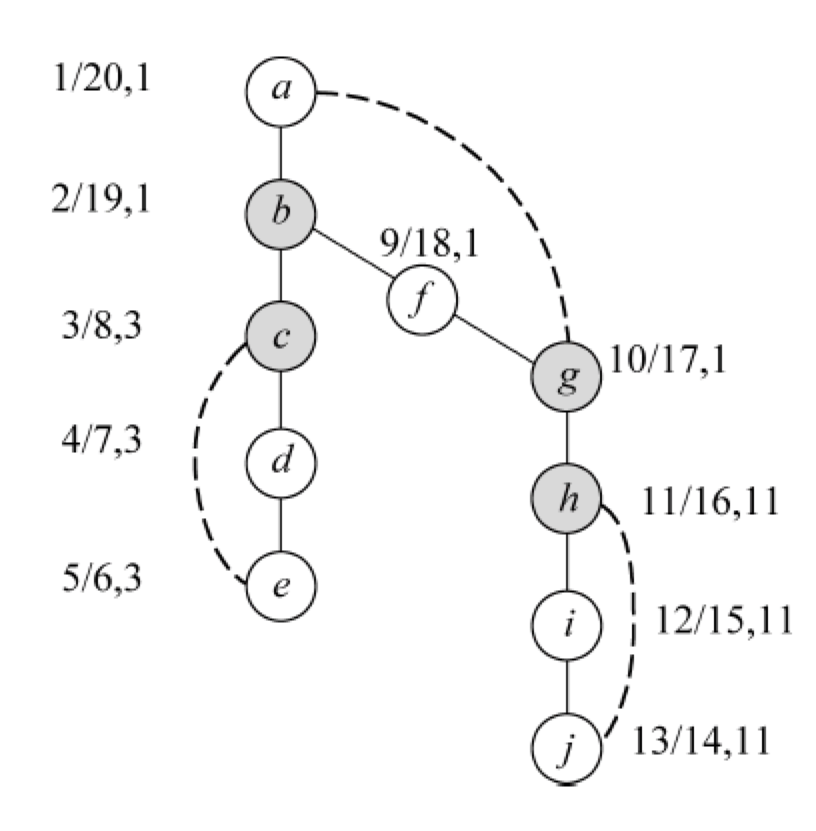


图 10-10　图 10-9(c)的深度优先树中各结点的 low 属性

3. 算法的伪代码描述

有了对无向连通图中关节点的上述观察，来设计一个基于 DFS 的计算无向连通图的关节点的算法。

```
ARTICULATION(G,s)
for each vertex u∈V[G]
    do color[u]←WHITE
       π[u]←NIL
time←rootdegree←0
A←S←∅
```

```
6 color[s]←GRAY
7 low[s]←d[s]←time←time+1
8 PUSH(S,s)
9 while S≠∅
10     do u←TOP(S)
11         for each v∈ Adj[u] and color[v]=GRAY
12             do low[u]=min{low[u],d[v]}
13                 if ∃v∈ Adj[u] and color[v]=WHITE
14                     then color[v]←GRAY
15                         π[v]←u
16                         low[v]←d[v]←time←time+1
17                         if u=s
18                             then rootdegree←rootdegree+1
19                         PUSH(S,v)
20                     else color[u]←BLACK
21                         POP(S)
22 for v←1 to n
23     do if π[v]≠NIL ▷非根节点
24         then if low[π[v]]>low[v]
25             then low[π[v]]←low[v]
26 if rootdegree>1
27     then 将 s 插入 A ▷根结点是关节点
28 for u←1 to n
29     do if π[u]≠s
30         then if low[u]≥d[π [u]]
31             then 将 π [u]插入 A ▷非根结点 π [u]是关节点
32 return A
```

算法 10-11　计算无向连通图的关节点的算法

在此过程中,需要计算每个顶点 u 的 low 属性,而 low 属性需要根据发现时间 d 和顶点间的父子关系 π 来计算。注意,实际上顶点 u 的发现时间和完成时间没有计算上的关系,所以可以省略完成时间而不影响发现时间和 low 属性的计算。

由于已知 G 是连通的,且指定了源顶点 s,所以,可以省略一般的 DFS 中的主循环。第 6 行和第 7 行将 s 的颜色发现时间和 low 属性初始化后,第 8 行将其压栈。第 9～21 行的 **while** 循环完成对 G 的 DFS。

对每一个顶点 v,压栈时将属性 $low[v]$初始化为发现时间 $d[v]$(见第 7 行和第 16 行)。在搜索 u 的邻接表过程中,一旦发现一条回边(我们知道,在无向图中,除了树枝边只有回边[①])(u,v),就将 $low[u]$置为 $low[u]$和 $d[v]$的较小者(见第 11 行和第 12 行)。在完成深度优先搜索后,第 22～25 行利用计算所得的 π(深度优先树中结点间的父子关系)数组,检测各顶点 $v(=\pi[u])$与其孩子顶点 u 之间 low 属性大小,将较小者赋予 $low[u]$。这样,对于每一个顶点,都按式(10-2)计算了它的 low 属性。

① 见本章 10.1.3 节的相关讨论。

对于源顶点 s，它作为优先树的根结点，一旦发现它有一棵非空子树，就将其子树计数器 *rootdegree* 增加 1(见第 17 行和第 18 行)。

如果根结点有多于 1 棵子树，则它是一个关节点(第 26 行和第 27 行)。对于非根结点 v，第 28～31 行检测是否有它的孩子结点 u 使得 $low[u] \geq d[v]$，若是，则 v 为一个关节点。

由于过程 ARTICULATION 是基于 DFS 的计算无向连通图的关节点，所以其时间复杂度和 DFS 的一样，为 $\Theta(V+E)$。

10.3.2　程序实现

下面在 C 语言中实现计算无向连通图的关节点算法。

```
1 LinkedList *articPoint(Graph *g,int s){
2     Stack *S=createStack(sizeof(int));
3     int n=g->n,u,v,time=0,rootDegree=0;
4     Color *color=(Color*)malloc(sizeof(Color));
5     int *pi=(int*)malloc(n*sizeof(int)),
6        *d=(int*)malloc(n*sizeof(int)),
7        *low=(int*)malloc(n*sizeof(int));
8     LinkedList *a=createList(sizeof(int),NULL);
9     ListNode **pos=(ListNode**)malloc(n*sizeof(ListNode*));
10   for(u=0;u<n;u++){
11     pi[u]=-1;
12     color[u]=WHITE;
13     pos[u]=g->adj[u]->nil->next;
14   }
15     color[s]=GRAY;
16     low[s]=d[s]=++time;
17     push(S,&s);
18     while(!stackEmpty(S)){
19       ListNode *p;
20       u=*(int*)(S->top->key);
21         p=pos[u];
22         v=(p!=g->adj[u]->nil)?((vertex*)(p->key))->index:-1;
23         while(p!=g->adj[u]->nil&&color[v]!=WHITE){
24             if(pi[u]!=v){                          /*回边*/
25                 if(low[u]>d[v])
26                     low[u]=d[v];
27             }
28             p=p->next;
29             v=(p!=g->adj[u]->nil)?((vertex*)(p->key))->index:-1;
30         }
31         pos[u]=p;
32         if(pos[u]!=g->adj[u]->nil){            /*找到白色顶点 v 并将其压入栈 S*/
33             if(u==s)                           /*u 是根*/
```

```
34                  rootDegree++;
35              color[v]=GRAY;
36              low[v]=d[v]=++time;
37              pi[v]=u;
38              push(S,&v);
39          }else{                          /* 否则没有与 u 邻接的顶点,则完成对 u 的访问 */
40              color[u]=BLACK;
41              pop(S);
42          }
43      }
44      for(u=0;u<n;u++){
45          if(pi[u]!=-1)
46              if(low[pi[u]]>low[u])
47                  low[pi[u]]=low[u];
48      }
49      if(rootDegree>1)
50          listPushBack(a,&s);
51      for(u=0;u<n;u++){
52          if(pi[u]!=s&&pi[u]!=-1){
53              if(low[u]>=d[pi[u]])
54                  listPushBack(a,pi+u);
55          }
56      }
57      return a;
58 }
```

程序 10-9　实现计算无向连通图的关节点的算法 10-11 的 C 源代码

由于在无向图中进行 DFS 所生成的深度优先树中除了树枝边就是回边,所以第 23～30 行对顶点 u 的邻接表扫描时,只要没遇到树枝边,遇到的就可能是回边(扫描到的顶点 v 是灰色的)。但是,即使相邻顶点 v 是灰色的,(u,v) 也未必是一条回边:在无向图中,(u,v) 和 (v,u) 是同一条边,如果这条边是按 v 到 u 顺序首次被访问到,那么,从 u 再次遇到 v,(u,v) 就不是一条回边。所以,对于扫描到灰色顶点只有满足 $\pi[u]\neq v$,(u,v) 才是一条回边。这就是第 24 行测试回边的条件。除此之外,实现的代码与算法的伪代码十分接近,此处不再赘述。

把程序 10-9 的代码存储为 graph 目录中的 articpoint.h 和 articpoint.c 文件,以备重用。

10.3.3　应用——雌雄大盗

Bonnie and Clyde

As two icons of the Great Depression, Bonnie and Clyde represent the ultimate criminal couple. Stories were written, headlines captured, and films were made about the two bank robbers known as Romeo and Juliet in a getaway car.

The new generation of Bonnie and Clyde is no longer cold-blooded killers with guns. Due to the boom of internet, they turn to online banks and scheme to hack the safety system. The safety system consists of number of computers connected by bidirectional cables. Since time is limited, they decide that they will attack exactly two computers A and B in the network, and as a result, other computers won't be able to transmit messages via A and B. The attack is considered successful if there are at least two computers (other than A and B) that disconnected after the attack.

As they want to minimize the risk of being captured, they need to find the easiest way to destroy the safety system. However, a brief study of the network indicates that there are many ways to achieve their objective; therefore they kidnapped the computer expert, you, to help with the calculation. To simplify the problem, you are only asked to tell them how many ways there are to destroy the safety system.

Input

There are multiple test cases in the input file. Each test case start with two integers N ($3 \leqslant N \leqslant 1000$) and M ($0 \leqslant M \leqslant 10000$), followed by M lines describing the connections between the N computers. Each line contains two integers A, B ($1 \leqslant A, B \leqslant N$), which indicates that computer A and B are connected by a bidirectional cable.

There is a blank line between two successive test cases. A single line with $N=0$ and $M=0$ indicates the end of input file.

Output

For each test case, output one integer number representing the ways to destroy the safety system in the format as indicated in the sample output.

Sample Input	Output for the Sample Input
4 4 1 2 2 3 3 4 4 1 7 9 1 2 1 3 2 3 3 4 3 5 4 5 5 6 5 7 6 7 0 0	Case 1: 2 Case 2: 11

1. 问题描述与分析

美国大萧条时代的象征,人称逃亡路上的罗密欧与朱丽叶的银行大盗 Bonnie 和 Clyde 在因特网蓬勃发展的今天又得到了新生。他们一改冷血杀手的旧面貌,干起了网上银行的盗窃行当。他们打算破坏网上银行的安全系统。网上银行的安全系统由若干台双向连接的计算机组成。将系统中的计算机视为顶点,连接两台计算机的通信线路视为顶点间的边,则该安全系统就抽象成一个无向图(因为每一条边都是双向的)。问题要求对每一个给定的无向连通图(安全系统),计算有多少种方法从中删除两个顶点,而使得图不再连通(其余各顶点中至少有两个互不可达)。这和计算无向连通图的关节点相似,不过计算的是删除两个顶点导致不连通的情形。问题形式化为如下。

输入:无向连通图 G。

输出:从 G 中删除两个顶点,使得 G 不连通的方法数。

若图 G 中存在关节点,则关节点与图中任一其他顶点均可构成破坏系统的点对。当 G 中无关节点时,可以通过检测所有 $\binom{N}{2}=N(N-1)/2$ 对顶点被删除后的 $N(N-1)/2$ 个子图的连通性来解决此问题。

2. 算法描述

```
BONNIE-CLYDE(G)
n←|V[G]|
A←ARTICULATION(G,s) ▷s 是 G 中任一顶点
m←length[A]
ways←0
if m≠0
   then for i←1 to m
            do ways←ways+(n-i)
return ways
for each u,v∈V[G]
   do G'←DELETE-VERTEX(G,u)
      G'←DELETE-VERTEX (G',v-1)
      if DFS(G')后得到多于 1 棵树的森林
         then ways←ways+1
return ways
```

算法 10-12 解决 Bonnie and Clyde 问题的过程

其中的 DELETE-VERTEX(G,u)过程,是在 G 中删除顶点 u 返回所得子图。可用伪代码描述如下。

```
DELETE-VERTEX (G,u)
n←|V[G]|
for v←1 to u-1
   do Adj'[v]←Adj[v]
```

```
4 for v←u+1 to n
5   do Adj'[v−1]←Adj[v]
6 for v←1 to n−1
7   do for each w∈Adj'[v]
8     do if w>u
9       then w←w−1
10      else if w=u
11        then 从 Adj'[v]中删除 w
12 return G'
```

算法 10-13　在图中删掉一个顶点的过程

过程运行时，第 2 行和第 3 行、第 4 行和第 5 行两个 **for** 循环完成在顶点邻接表数组中删除顶点 u 的邻接表。第 6～11 行的 **for** 循环对剩下的每一个顶点的邻接表扫描，对表中的结点做如下操作：编号等于 u 的，直接删除；编号小于 u 的，保留不变；编号大于 u 的，编号减 1。

3. 程序实现

1）在图中删除一个顶点

在编写解决该问题的程序之前，先来实现在图中删除一个顶点的算法 10-13。

```c
void deleteVertex(Graph *g,int u){
 int v,n=g->n-1;
 LinkedList **adj=(LinkedList**)malloc(n*sizeof(LinkedList*));
 memcpy(adj,g->adj,u*sizeof(LinkedList*));
 memcpy(adj+u,g->adj+u+1,(n-u)*sizeof(LinkedList*));
 for(v=0;v<n;v++){
   ListNode *p=adj[v]->nil->next;
   while(p!=adj[v]->nil){
     if(((vertex*)p->key)->index==u){
         ListNode *q=p->next;
         listDelete(adj[v],p);
         clrListNode(p,NULL);
         free(p);p=q;
         continue;
      }
      if(((vertex*)p->key)->index>u)
         ((vertex*)p->key)->index--;
      p=p->next;
    }
  }
  free(g->adj);g->adj=adj;
  g->n=n;
}
```

程序 10-10　在图中删除一个顶点的 C 函数

与算法稍有不同,函数中删除顶点 u 的操作是针对参数 g 进行的,所以不必返回任何数据。第 4 行和第 5 行调用库函数 memcpy 将 g 的 adj[0..u－1]及 adg[u＋1..n－1]直接复制给目标数组,实现算法过程中的第 2 行和第 3 行及第 4 行和第 5 行的两个 **for** 循环。第 6～19 行的 for 循环完成算法过程中第 6～11 行的操作。第 21 行和第 22 行将删除顶点 u 后的数据更新 g。

为便于代码重用,将程序 10-10 添加到文件夹 graph 中的头文件 graph.h 和源文件 graph.c 中。

2)**实现程序**

```
1 int numOfTree(Graph *g){
2  pair r,*df;
3  int *pi,i,nt=0,*d,*f;
4  r=dfs(g);
5  pi=(int*)r.first;
6  d=(int*)(((pair*)r.second)->first);
7  f=(int*)(((pair*)r.second)->second);
8  for(i=0;i<g->n;i++)
9    if(pi[i]==-1)
10       nt++;
11  free(pi);free(d);free(f);free(r.second);
12  return nt;
13 }
14 int main(){
15  int n,m,count=0;
16  FILE *f1,*f2;
17  assert(f1=fopen("chap10/Bonnie and Clyde/inputdata.txt","r"));
18  assert(f2=fopen("chap10/Bonnie and Clyde/outputdata.txt","w"));
19  fscanf(f1,"%d%d",&n,&m);
20  while(n||m){                              /*处理每一个案例*/
21    Graph *g=zeroGraph(n);
22    LinkedList *atps;
23    int i,ways=0,number;
24    for(i=0;i<m;i++){
25      int u,v;
26      fscanf(f1,"%d%d",&u,&v);
27      addEdge(g,u-1,v-1,1.0);
28      addEdge(g,v-1,u-1,1.0);
29    }
30    atps=articPoint(g,0);
31    number=atps->n;clrList(atps,NULL);free(atps);
32    if(number)
33      for(i=1;i<=number;i++)
34        ways+=(n-i);
```

```
35      else{
36          int u,v;
37          for(u=0;u<n;u++)
38              for(v=u+1;v<n;v++){
39                  int k, * pi;
40                  Graph  * g1=cloneGraph(g);
41                  deleteVertex(g1,u);
42                  deleteVertex(g1,v-1);
43                  if(numOfTree(g1)>1)
44                      ways++;
45                  graphClear(g1);free(g1);
46              }
47      }
48      fprintf(f2,"Case %d: %d\n",++count,ways);
49      graphClear(g);free(g);
50      fscanf(f1,"%d%d",&n,&m);
51  }
52  fclose(f1);fclose(f2);
53  return 0;
54 }
```

程序 10-11　解决 Bonnie and Clyde 问题的 C 程序

对程序 10-11 的说明如下。

（1）第 20～51 行的 **while** 循环处理输入文件中的每一个案例。其中，第 21 行创建了一个具有 n 个顶点的零图 g，第 23 行声明了一个存储计算结果的变量 ways，并初始化为 0。

（2）第 24～29 行的 **for** 循环在输入文件中逐一读取顶点对 u、v，并将边(u，v)和(v，u)插入图 g(这是因为 g 是无向图)。

（3）第 30～51 行代码实现算法 10-13。第 30 行调用程序 10-9 定义的函数 articPoint，计算图 g 的关节点。第 31 行解析出关节点的个数 number。第 32～34 行对具有关节点的情形加以处理，而 35～46 行处理 g 中无关节点的情形。其中，第 37～46 行的两重 **for** 循环实现算法 10-13 中第 6～11 行的操作，对在图 g 中删除一对顶点得到的 n(n－1)/2 个图 g1 逐一检验是否具有连通性。累计不具有连通性的情形数即为所求。其中，第 43 行调用的函数 numOfTree 计算 g1 的连通分支数，该函数定义于第 1～13 行。函数中调用 dfs 对 g1 进行深度优先搜索，对返回的的数组 pi 计算其中值为－1 的元素个数，即深度优先森林中所含树的棵树，由于 g1 是无向图，所以深度优先森林中树的棵树就是连通分支个数。

程序 10-11 存储在文件夹 chap10/Bonnie and Clyde 中的源文件 Bonnie andClyde. c 中，读者可打开文件研读，并试运行。

10.4 广度优先搜索

10.4.1 算法描述与分析

1. 问题的理解与描述

广度优先搜索(Broad First Search,BFS)是图搜索的一种最简单的算法,并且也是很多重要图算法的原型。给定一个图(有向或无向)$G=\langle V,E\rangle$和其中的一个**源**顶点 s,广度优先搜索系统地探索 G 的边以"发现"从 s 出发每一个可达的顶点:发现从 s 出发距离为 $k+1$ 的顶点之前先发现距离为 k 的顶点。搜索所经路径中的顶点,按先后顺序构成"父子关系":先发现的顶点 u,并由 u 出发发现与其相邻的顶点 v,则称 u 为 v 的父亲。由于每个顶点只有最多一个顶点作为它的父亲,所以搜索路径必构成一棵根树(树根为起始顶点 s)G_π,把这棵树称为 G 的广度优先树。与此同时,还计算出了从 s 到这些可达顶点的距离(最少的边数即"最短路径")。这样,图的广度搜索问题形式化描述如下。

输入:图 $G=\langle V,E\rangle$,源顶点 $s\in V$。

输出:G 的广度优先树 G_π 以及树中从树根 s(源顶点)到各结点的距离。

2. 算法的伪代码描述

为跟踪整个过程,广度优先搜索为每个顶点着白色、灰色或黑色。开始时,所有的顶点都着白色,然后可能变成灰色,再后来可能变为黑色。一个顶点在搜索过程中首次被遇到称为**被发现**,此后它就不再是白色的了。所以,灰色的或黑色的顶点是已经发现的,广度优先搜索用两者间的区别来保证搜索进程以广度优先的方式进行。若$(u,v)\in E$ 且顶点 u 是黑色的,则顶点 v 不是灰色就是黑色的,即与黑色顶点相邻的顶点必是已访问过的。灰色顶点可能有白色相邻的顶点;它们表示已访问过的与未访问过的顶点之间的界线。

过程 BFS 假定输入的图 G 是用邻接表表示的。每个顶点 $u\in V$ 的颜色存储在 $color[u]$ 中。为计算图 G 的广度优先树 G_π 和从 s 到各可达顶点的距离,用 $\pi[u]$表示顶点 u 在 G_π 中的父结点,用 $d[u]$表示从 s 到 u 的距离。与 DFS 使用先进后出的栈不同,算法使用一个先进先出的队列 Q 来管理灰色顶点集合。

```
BFS(G,s)
for 每个顶点 u∈V[G]-{s}
    do color[u]←WHITE
       d[u]←∞
       π[u]←NIL
color[s]←GRAY
d[s]←0
Q←∅
ENQUEUE(Q,s)
while Q≠∅
```

```
10     do u←DEQUEUE(Q)
11         for each v ∈Adj[u]
12             do if color[v]=WHITE
13                 then color[v]←GRAY
14                     π[v]←u
15                     d[v]←d[u]+1
16                     ENQUEUE(Q,v)
17         color[u]←BLACK
18 return π and d
```

算法 10-14　图的广度优先搜索算法

第 1～7 行进行初始化工作：将所有顶点 u 的颜色属性 $color[u]$置为 WHITE(未被访问)，距离属性置 $d[u]$置为∞，父结点指针 $\pi[u]$置为 NIL。将入队的源顶点 s 的颜色属性 $color[s]$置为 GRAY(开始访问)，距离属性 $d[s]$置为 0。第 8 行将 s 加入队列 Q。

第 9～15 行的 **while** 循环重复执行直至队列 Q 空为止。每次重复将队首元素 u(某灰色顶点)出队，将图 G 中所有与 u 邻接的白色顶点 v 着成灰色(开始访问)，将 v 的父结点指针指向 u，并将 v 的距离属性 $d[v]$置为 $d[u]+1$，即从 s 到 v 的距离置为 s 到其父结点 u 的距离加 1，然后入队。最后将 u 着成黑色(完成访问)。

第 9～15 行的 **while** 循环只要还有灰色顶点就会重复，这些灰色顶点是已被发现但尚未扫描其邻接表的顶点。为证明此算法的正确性，需要说明算法运行结束时所有从 s 可达的顶点 v 都被搜索到，且 $\pi[v]$记录下从 s 到 v 的一条最短路径上 v 的父结点，$d[v]$是这条最短路径的长度。

图 10-11 示例了 BFS 作用在一个简单的图上的过程。由 BFS 产生的树中的边表示为带有阴影。每个顶点 u 内部显示的是 $d[u]$。队列 Q 显示的是第 8～15 行的 **while** 循环每次重复前的状态。队列中顶点距离显示在顶点下方。

3. 算法的正确性

引理 10-3　从源顶点 s 到任何顶点 v 的距离必不超过运行 BFS 后过此顶点的 d 属性。

这是因为，若 v 从 s 不可达，则其 d 属性将维持初始值∞。若 v 从 s 可达，则在 BFS 过程中其 d 属性得到改写时必经过一条从 s 出发的路径，其长度恰为 $d[v]$。按照距离的意义，它是从 s 到 v 的最短路径的长度。所以，$d[v]$作为一条从 s 到 v 的路径长度当然不会小于s 到 v 的距离。

引理 10-4　设队列 $Q=\{v_1,v_2,\cdots,v_r\}$，则 $d[v_r]\leqslant d[v_1]+1$(即队尾元素 d 的属性不超过队首元素 d 的属性加 1)，且 $d[v_1]\leqslant d[v_2]\leqslant\cdots\leqslant d[v_r]$。

要证明这一事实，可以对算法中对队列 Q 的操作次数 k 做数学归纳。$k=1$ 时，即对 Q 的第 1 次操作，发生在第 8 行。此时 $Q=\{s\}$，结论当然为真。

假定 $k<i(i>1)$时，结论为真，即若 $Q=\{v_1,v_2,\cdots,v_r\}$，$d[v_r]\geqslant d[v_1]+1$，且 $d[v_1]\leqslant d[v_2]\leqslant\cdots\leqslant d[v_r]$。下面证名 $k=i(i>1)$时，结论也为真。这要对第 $k=i$ 次操作的种类分别讨论。首先，假定本次操作是第 10 行的出队操作。这时，原来的队首元素 v_1 被弹出队

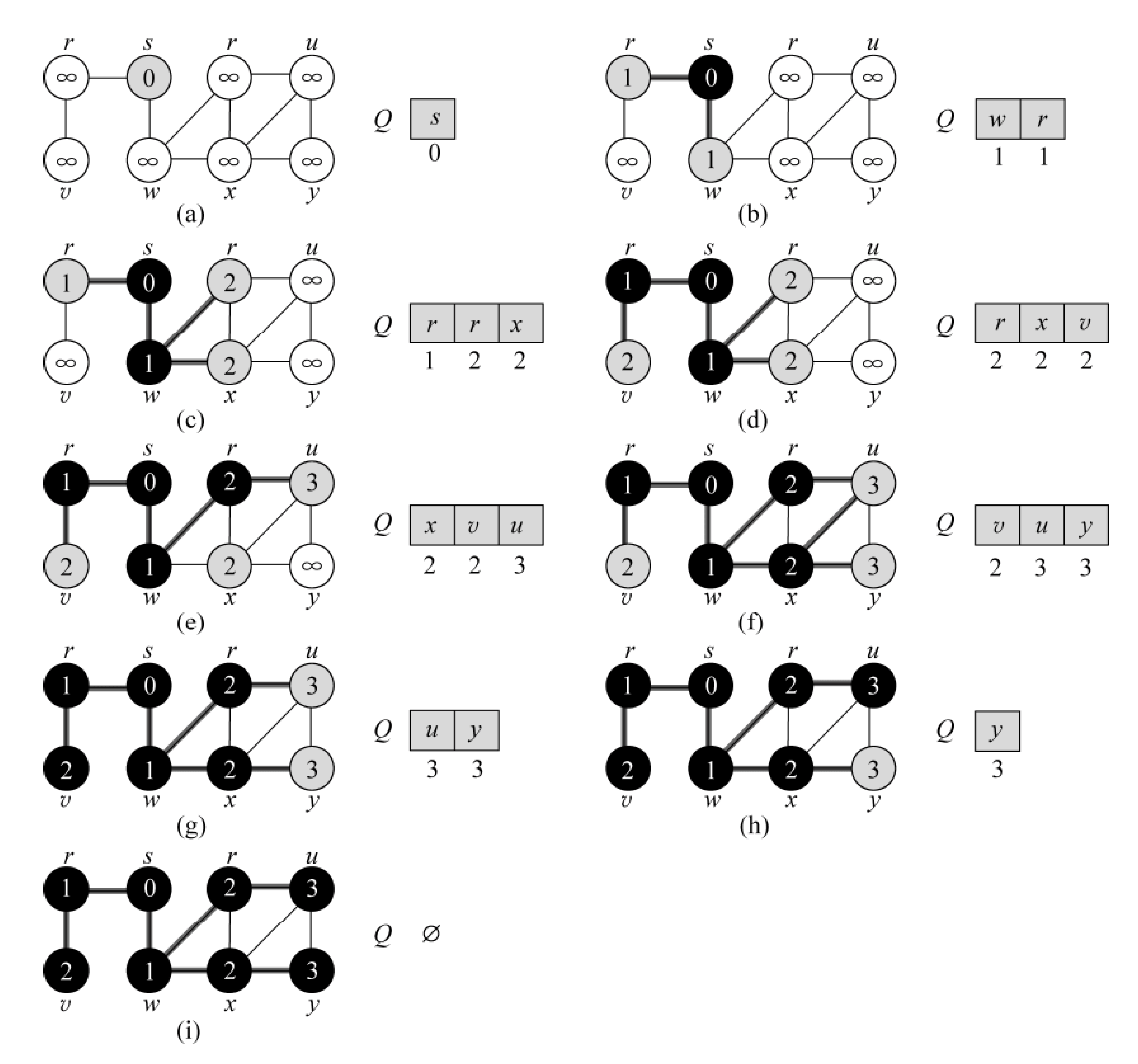


图 10-11　BFS 对一个无向图的操作

列,而原来的元素 v_2 成为队首。这时根据归纳假设,$d[v_r]\leqslant d[v_1]+1\leqslant d[v_2]+1$,即队尾元素的 d 属性不超过队首元素的 d 属性加 1。而不等式 $d[v_2]\leqslant\cdots\leqslant d[v_r]$继续保持。

其次,假定第 k 次操作是发生在第 16 行的入队操作。这里又需要分两种情况:其一,上次操作是出队操作,出队的顶点为 u。根据归纳假设得出 u 出队前 $d[u]=d[v_1]\leqslant d[v_2]\leqslant\cdots\leqslant d[v_r]\leqslant d[v_1]+1=d[u]+1$。$u$ 出队后,v_2 成为队首。本次入队的顶点 v 在第 15 行获得新的 d 属性 $d[v]=d[u]+1\leqslant d[v_2]+1$,即 v 入队后,作为队尾元素的 d 属性不超过队首元素的 d 属性加 1。此外,$d[v_2]\leqslant\cdots\leqslant d[v_r]\leqslant d[v_1]+1=d[u]+1=d[v]$,即队列中所有顶点的 d 属性不减。其二,上次操作也是入队操作,所以,本次的入队顶点和上次入队的顶点都与顶点 u 相邻,所以,它们的 d 属性相等都等于 $d[u]+1$。至此,本引理得证。

由引理 10-4 知,在 BFS 过程中,顶点按入队的前后顺序,其 d 属性不减,即先入队的顶点的 d 属性不会超过后入队的顶点的 d 属性。

利用引理 10-3 和引理 10-4,可以说明引理 10-5。

引理 10-5　运行 BFS 后,图 G 中各顶点 v 的 d 属性记录了 s 到 v 的距离。

对 s 到 v 的距离 k 做数学归纳。$k=1$ 时,即考虑所有与 s 相邻的顶点 v,它们在 BFS 的第 9~18 行的 **while** 循环的第 1 次重复中均被染成灰色且将 d 属性置为 1 后入队,由于在 BFS 中,每个顶点至多进入队列 Q 一次,所以这些顶点的 d 属性此后不会被改变,即过程结束时,其 d 属性记录了从 s 到 v 的距离。

假定对所有从 s 出发距离为 $k=i(>1)$ 的顶点 v，运行 BFS 后，d 属性记录了距离，即 $d[v]=i$。下面证明对于所有从 s 出发距离为 $k=i+1$ 的顶点，在 BFS 中将得到值为 $i+1$ 的 d 属性值。设 v 是任一从 s 出发距离为 $i+1$ 的顶点，并设 $p: s \quad u \to v$ 为 s 到 v 的一条最短路径，即 $|p|=i+1$。设 p 中从 s 到顶点 u 的部分为 p_1，则 p_1 为 s 到 u 的一条最短路径(这是因为若否，则有 s 到 u 的一条更短路径，$p_1': s \quad u$，则 p_1' 连接(u,v)将得到一条从 s 到 v 的比 p 更短的路径，这与 p 是最短路径矛盾)。显然，$|p_1|=i$，根据归纳假设，$d[u]=i$。我们断言，u 出队时，v 是白色的。否则，v 比 u 更先入队，根据引理 10-3 知，$d[v]\geqslant i+1$；而根据引理 10-4 知，$d[u]\geqslant d[v]\geqslant i+1$。这与 $d[u]=i$ 矛盾。这样，u 出队时，第 11～17 行的 **for** 循环必能在第 13 行检测出 v 是与 u 相邻且为白色的，并在第 15 行得到它的 d 属性值为 $d[u]+1=i+1$。

最后，观察算法可知，对所有从 s 可达的顶点 v，$d[v]$的值是在第 14 行确定了 $\pi[v]$为顶点 u 后，第 15 行将其确定为其父结点 u 的 $d[u]$值加 1。根据上述 d 属性的说明可知记录在 $\pi[v]$中的 u 确实就是 s 到 v 的一条最短路径上 v 的父结点。

4. 算法的运行时间

第 1～4 行的循环重复$|V|$次。另一方面，由于每条边在搜索过程中有且仅有一次被访问，第 9～17 行两重循环嵌套内的操作被执行$|E|$次。所以 BFS 的总运行时间是 $\Theta(V+E)$。于是，广度优先搜索运行于 G 的邻接表示规模的线性时间内。

10.4.2　程序实现

由于在图的广度优先搜索算法 10-14 中要借助一个队列来管理各顶点的搜索顺序，所以我们还需要实现队列的数据结构。这在第 2 章中的程序 2-15 和程序 2-16 中已经定义了。

```
pair bfs(Graph *g,int s){
    Queue *Q=createQueue(sizeof(int));
    int *pi,*d,i,u,v,n=g->n;
    Color *color;
    pi=(int*)malloc(n*sizeof(int));
    d=(int*)malloc(n*sizeof(int));
    color=(Color*)malloc(n*sizeof(Color));
    for(i=0;i<g->n;i++){
        pi[i]=-1;
        d[i]=INT_MAX;
        color[i]=WHITE;
    }
    d[s]=0;
    color[s]=GRAY;
    enQueue(Q,&s);
    while(!queueEmpty(Q)){
        ListNode *q=deQueue(Q);
        u=*(int*)q->key;
        clrListNode(q,NULL);
        q=g->adj[u]->nil->next;
```

```
21          while(q!=g->adj[u]->nil){
22            v=((vertex *)(q->key))->index;
23               if(color[v]==WHITE){
24                    color[v]=GRAY;
25                    d[v]=d[u]+1;
26                    pi[v]=u;
27                    enQueue(Q,&v);
28               }
29               q=q->next;
30          }
31          color[u]=BLACK;
32     }
33     clrQueue(Q,NULL);
34     free(Q);
35     return make_pair(pi,d);
36 }
```

程序 10-12　实现图的广度优先搜索算法 10-14 的 C 源代码

对程序 10-12 的说明如下。

(1) bfs 的两个参数分别是图的邻接表表示 g 和源顶点 s。由于 BFS 要返回两个数组,所以在 C 中需要将它们封装成一个对象。利用在第 8 章的 8.3.4 节中开发的结构体pair 来作为封装两个数组的工具。

(2) 第 2 行声明了一个队列 Q。第 3 行声明的整型指针 d 和 pi 对应于算法中的数组 d 和 π,u 和 v 对应于算法中同名的临时表示顶点的变量。第 4 行声明了一个与算法中同名的顶点颜色数组 color。第 8～15 行实现算法中第 1～8 行的初始化工作。

(3) 第 16～32 行实现的是算法第 9～17 行的 **while** 循环。由于在 C 中为了让队列这一数据结构具有通用性,使用了 **void** * 指针,所以程序中为了使用这样的数据结构就要借助指针来完成对数据的读写。读者需要对代码中的这些指针操作给予充分的注意。第 35 行将数组 pi 和 d 封装于 pair 对象中并将其返回。

为便于代码重用,将程序 10-12 存储在文件夹 graph 中的头文件 bfs.h(函数原型声明)和源文件 bfs.c(函数定义代码)中。

10.4.3　应用——攻城掠地

Risk

Risk is a board game in which several opposing players attempt to conquer the world. The gameboard consists of a world map broken up into hypothetical countries. During a player's turn, armies stationed in one country are only allowed to attack only countries with which they share a common border. Upon conquest of that country, the armies may move into the newly conquered country.

During the course of play, a player often engages in a sequence of conquests with the

goal of transferring a large mass of armies from some starting country to a destination country. Typically, one chooses the intervening countries so as to minimize the total number of countries that need to be conquered. Given a description of the gameboard with 20 countries each with between 1 and 19 connections to other countries, your task is to write a function that takes a starting country and a destination country and computes the minimum number of countries that must be conquered to reach the destination. You do not need to output the sequence of countries, just the number of countries to be conquered including the destination. For example, if starting and destination countries are neighbors, then your program should return one.

The following connection diagram illustrates the sample input.

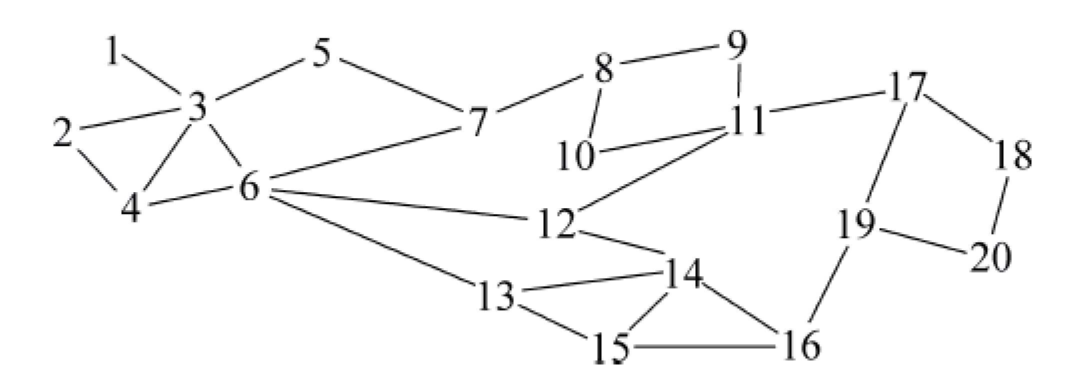


图 10-12　连接图

Input Format

Input to your program will consist of a series of country configuration test sets. Each test set will consist of a board description on lines 1 through 19. The representation avoids listing every national boundary twice by only listing the fact that country I borders country J when $I<J$. Thus, the Ith line, where I is less than 20, contains an integer X indicating how many "higher-numbered" countries share borders with country I, then X distinct integers J greater than I and not exceeding 20, each describing a boundary between countries I and J. Line 20 of the test set contains a single integer ($1 \leqslant N \leqslant 100$) indicating the number of country pairs that follow. The next N lines each contain exactly two integers ($1 \leqslant A, B \leqslant 20$; $A \neq B$) indicating the starting and ending countries for a possible conquest.

There can be multiple test sets in the input; your program should continue reading and processing until reaching the end of file. There will be at least one path between any two given countries in every country configuration.

Required Output Format

For each input set, your program should print the following message "Test Set #T" where T is the number of the test set starting with 1. The next NT lines each will contain the result for the corresponding test in the test set-that is, the minimum number of countries to conquer. The test result line should contain the start country code A, the string "to", the destination country code B, the string ":" and a single integer indicating the minimum number of moves required to traverse from country A to country B in the test set. Following all result lines of each input set, your program should print a single blank line.

Sample Input

1 3
2 3 4
3 4 5 6
1 6
1 7
2 12 13
1 8
2 9 10
1 11
1 11
2 12 17
1 14
2 14 15
2 15 16
1 16
1 19
2 18 19
1 20
1 20
5
1 20
2 9
19 5
18 19
16 20

Sample Output

Test Set ＃1
1 to 20：7
2 to 9：5
19 to 5：6
18 to 19：2
16 to 20：2

1. 问题描述与分析

Risk 是一款战略游戏。模拟世界的沙盘分成 20 个城市，每个城市与其他 1～19 个城市相连。任一玩家要从一个指定的起始城市进军到另一个指定的目标城市。轮到己方玩时，玩家需从当前占领的城市攻击下一个与之相连的城市。这样，他就可以让其军团占领这个城市。当然，每攻下一个城市，就要分兵占领。所以，希望计算从起始城市到目标城市之间最少需要攻占的城市数。如果将城市视为顶点，相连的城市表示为顶点间的边，这将构成一个无向无权图 G。本问题是对该无向图 G 中指定的源顶点 u 和目标顶点 v 之间计算最短路径(路径所含边数最少)所含顶点数(包括目标顶点 v，但不包括源顶点 u)的问题。这恰好

是 u、v 之间最短路径的长度。所以，可以利用 BFS，对给定的无向图 G 中顶点 u、v，调用 BFS(G,u)，利用返回的数组 d 计算所要求的数值 $d[v]$。

2. 程序实现

利用 BFS 的实现函数，很容易解决 Risk 问题。

```
1 int main(){
2  FILE *f1,*f2;
3  assert(f1=fopen("chap10/Risk/inputdata.txt","r"));
4  assert(f2=fopen("chap10/Risk/outputdata.txt","w"));
5  while(!feof(f1)){
6    int i,j,k,x,*d,n,A,B,c=0;
7    Graph *g=zeroGraph(20);
8    pair p;
9    for(i=0;i<19;i++){
10       fscanf(f1,"%d",&x);
11       for(k=0;k<x;k++){
12          fscanf(f1,"%d",&j);
13          j--;
14          addEdge(g,i,j,1.0);
15          addEdge(g,j,i,1.0);
16       }
17    }
18    fprintf(f2,"Test Set #%d\n",++c);
19    fscanf(f1,"%d",&n);
20    for(i=0;i<n;i++){
21       fscanf(f1,"%d%d",&A,&B);
22       p=bfs(g,A-1);
23       free(p.first);d=(int*)p.second;
24       fprintf(f2,"%d to %d: %d\n",A,B,d[B-1]);
25       free(d);
26    }
27    graphClear(g);free(g);
28  }
29  fclose(f1);fclose(f2);
30  return 0;
31 }
```

程序 10-13　解决 Risk 问题的 C 程序

对程序 10-13 的说明如下。

(1) 第 5～28 行的 **while** 循环处理输入文件中的每一个案例。其中，第 7 行创建了一个具有 20 个顶点的零图 g。第 9～17 行的嵌套 **for** 循环将案例数据中每一个城市与其他城市的相邻信息，作为图中的边插入 g。

(2) 第 20～26 行的 **for** 循环对案例中每一对起止点 A、B，通过第 22 行调用函数

bfs,计算出从 A 出发到达其他城市的最短距离,第 24 行将 A 到 B 的最短距离写入输入文件。

程序 10-14 存储为文件夹 chap10/Risk 中的源文件 risk.c,读者可打开文件研读,并试运行。

10.5 流网络与最大流问题

10.5.1 算法描述与分析

1. 问题理解与描述

1) **流网络**

一个**流网络** $G=<V,E>$是一个有向图,其中的每一条边$(u,v)\in E$有一个非负的实数**容量** $c(u,v)\geqslant 0$。若$(u,v)\notin E$,假定 $c(u,v)=0$,即:

$$c(u,v)=\begin{cases}\text{非负实数} & u,v\in V,\text{且}(u,v)\in E\\ 0 & u,v\in V,\text{且}(u,v)\notin E\end{cases}$$

在网络中标识两个顶点:一个称为**源点**,记为 s;一个称为**汇点**,记为 t。假定每一个顶点都位于从源点到汇点之间的某条路径上,即对每一个顶点 $v\in V$,有一条路径 s v t。所以,该图是连通的,且$|E|\geqslant|V|-1$。网络的一个例子,如图 10-13 所示。

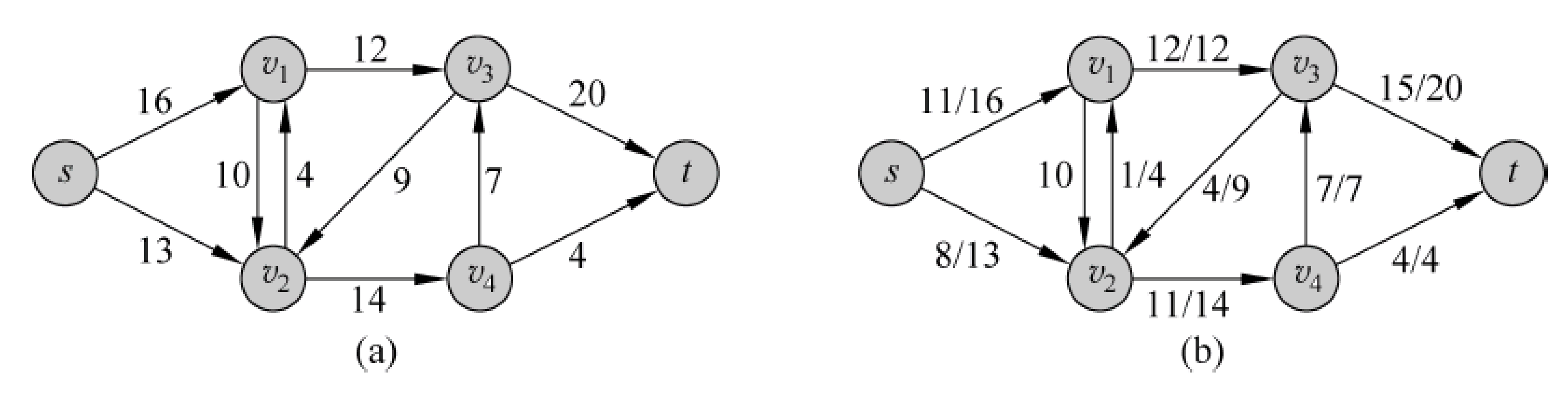


图 10-13 网络图

图 10-13(a)为网络 $G=(V,E)$。源点为 s,汇点为 t。每条边都标注其容量。图 10-13(b)中,G 中的一个流 f 其值$|f|=19$。图中只显示正的流,若 $f(u,v)>0$,边(u,v)标示为 $f(u,v)/c(u,v)$(斜杠仅用来分隔流与容量,并不表示除法)。若 $f(u,v)\leqslant 0$,边(u,v)仅标示为其容量。

2) **流**

设 $G=<V,E>$是一个网络,其容量函数为 c。设 s 为该网络的源点,t 是汇点。G 中的一个**流**是一个实数值函数 f:$V\times V\to\mathbf{R}$,它满足如下三条性质。

容量约束:对所有的 $u,v\in V$,要求 $f(u,v)\leqslant c(u,v)$。

斜对称性:对所有的 $u,v\in V$,要求 $f(u,v)=-f(v,u)$。

流守恒性:对所有的 $u\in V-\{s,t\}$,要求 $\sum_{v\in V}f(u,v)=0$。

量 $f(u,v)$可以是正的、零或负的,称为是从顶点 u 到顶点 v 的**流**。一个流 f 的**值**定义为

$$|f| = \sum_{v \in V} f(s,v) \tag{10-3}$$

也就是说，流出源点的总的流（此处，记号 $|\cdot|$ 表示流的值，并非绝对值或势）。

对于任何流网络 G，必存在流 f。例如，对所有的 $(u,v) \in V \times V$，令 $f(u,v)=0$，这一函数满足流的所有 3 个性质，所以它是 G 的一个流。把这一特殊的流记为 f_0。显然，$|f_0|=0$。

3）**最大流问题**

在**最大流问题**中，已知一个网络 G 及其源点 s 和汇点 t，希望找到具有最大值的流。形式化为如下。

输入：网络 $G=\langle V,E \rangle$，其中源点为 s，汇点为 t，定义在 $V \times V$ 上的容量 c。

输出：定义在 $V \times V$ 上的流 f：$V \times V \to \mathbf{R}$，使得 $|f| = \sum_{v \in V} f(s,v)$ 最大。

4）**剩余网络**

在解决最大流问题的方法中，有一个很重要的概念——剩余网络。直观地看，对给定的网络及一个流，其剩余网络由可以接受更多流的边组成。更形式化地说，假定有一个网络 $G=\langle V,E \rangle$，其源点为 s，汇点为 t。设 f 为 G 中的一个流，考虑一对顶点 $u,v \in V$。不超过容量 $c(u,v)$，从 u 到 v 可以添加的流量是 (u,v) 的**剩余容量**，定义如下：

$$c_f(u,v) = c(u,v) - f(u,v) \tag{10-4}$$

例如，若 $c(u,v)=16$ 且 $f(u,v)=11$，则可以在对边 (u,v) 的容量限制下对 $f(u,v)$ 增加 $c_f(u,v)=5$ 个单位。当流 $f(u,v)$ 为负时，剩余容量 $c_f(u,v)$ 将大于容量 $c(u,v)$。例如，若 $c(u,v)=16$ 且 $f(u,v)=-4$，则剩余容量 $c_f(u,v)$ 为 20。

给定一个流网络 $G=\langle V,E \rangle$ 和一个流 f，G 的根据 f 的**剩余网络**记为 $G_f=\langle V,E_f \rangle$，其中

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

也就是说，剩余网络的每一条边，也称为**剩余边**，可容纳一个大于 0 的流。

5）**增广路径**

要用到的另一个重要概念是流网络中的增广路径。为研究增广路径及其性质，先讨论流网络中两个流的和。已知流网络 $G=\langle V,E \rangle$，设 f_1 和 f_2 为从 $V \times V$ 到 $\mathbf{R}$ 的流函数。定义 $V \times V$ 到 $\mathbf{R}$ 的函数 f_1+f_2：

对任意的 $u,v \in V$，$(f_1+f_2)(u,v)=f_1(u,v)+f_2(u,v)$。

若 f_1+f_2 满足容量约束性，即对所有的 $u,v \in V$，有 $(f_1+f_2)(u,v) \leqslant c(u,v)$，不难得知，$f_1+f_2$ 是流网络 G 的一个流。这是因为 f_1+f_2 除了满足容量约束性外，对所有的 $u,v \in V$，有 $(f_1+f_2)(u,v)=f_1(u,v)+f_2(u,v)=-f_1(v,u)+f_2(v,u)=-(f_1+f_2)(v,u)$，即 f_1+f_2 满足流的斜对称性。此外，对所有的 $u \in V-\{s,t\}$，有 $\sum_{v \in V}(f_1+f_2)(u,v) = \sum_{v \in V}(f_1(u,v)+f_2(u,v)) = \sum_{v \in V} f_1(u,v) + \sum_{v \in V} f_2(u,v) = 0+0 = 0$，即 f_1+f_2 满足流的守恒性质。

给定网络 $G=(V,E)$ 及一个流 f，一条**增广路径** p 是剩余网络 G_f 中的一条从 s 到 t 的简单路径。根据剩余网络的定义，增广路径上的每一条边 (u,v) 将接纳一些从 u 到 v 的附加正流而不会违背该边上的容量限制。

图 10-14(b) 中带有阴影的路径就是一条增广路径。将图中的剩余网络 G_f 视为一个网

络,可以对此路径上的每一条边增加 4 个单位的流而不违背容量限制,这是因为此路径上的最小剩余容量为 $c_f(v_2,v_3)=4$。我们称能在增广路径 p 的每条边上增加的最大流量为 p 的**剩余容量**,定义如下:

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ 在 } p \text{ 上}\}$$

设 $G=(V,E)$ 为一网络,f 为 G 中的一个流,p 为 G_f 中的一条增广路径。定义函数 f_p: $V\times V \to \mathbf{Z}$ 为

$$f_p(u,v) = \begin{cases} c_f(p) & (u,v) \text{ 在 } p \text{ 上} \\ -c_f(p) & (v,u) \text{ 在 } p \text{ 上} \\ 0 & \text{其他} \end{cases} \tag{10-5}$$

则可以验证 f_p 是 G_f 中的一个值为 $|f_p|(|f_p|=c_f(p)>0)$ 的流。首先,对于 p 上的每一条边 (u,v),由 $f_p(u,v)=c_f(p)=\min\{c_f(u,v) : (u,v) 在 p 上\}$ 知,对 p 上的每一条边 (u,v),有 $f_p(u,v)\leqslant c_f(u,v)$,而对 G_f 中的所有其他边 (u,v),$f_p(u,v)=0\leqslant c_f(u,v)$,即 f_p 满足流的容量约束性。其次,对于所有 p 上的每一条边 (u,v),根据式(10-4)可知 $f_p(u,v)=-f_p(u,v)$,而对 G_f 中的所有其他边 (u,v),$f_p(u,v)=0=-f_p(v,u)$,即 f_p 满足流的斜对称性。最后,对所有的 $u\in V-\{s,t\}$,$v\in V$,若 (u,v) 在 p 上,则可能有 3 种情形:其一,$v=s$,此时 $f_p(u,v)=-f_p(v,u)=-f_p(s,u)=-c_f(p)$;其二,$v\neq s$ 且 $v\neq t$,此时,$f_p(u,v)$ 和 $f_p(v,u)$ 都出现在和式 $\sum_{v\in V} f_p(u,v)$ 中,根据斜对称性相互抵消;其三,$v=t$,此时 $f_p(u,v)=f_p(u,t)=c_f(p)$,它与第一种情形相加亦相互抵消。对 (u,v) 和 (v,u) 均不在 p 上的情形 $f_p(u,v)=0$。总之对所有的 $u\in V-\{s,t\}$,$v\in V$,$\sum_{v\in V} f_p(u,v) = 0$。此即说明 f_p 是 G_f 的一个流,而 $|f_p|=c_f(p)$ 是因为增广路上所有边的流量都是 $c_f(p)$。

图 10-14(a)为图 10-13(b)中的流网络 G 和流 f。图 10-14(b)剩余网络 G_f,带有阴影的是增广路径 p;其剩余容量为 $c_f(p)=c(v_2,v_3)=4$ 。图 10-14(c)为沿路径 p 增加剩余容量4 所得的 G 中的流。图 10-14(d)为由图 10-14(c)中的流引出的剩余网络。

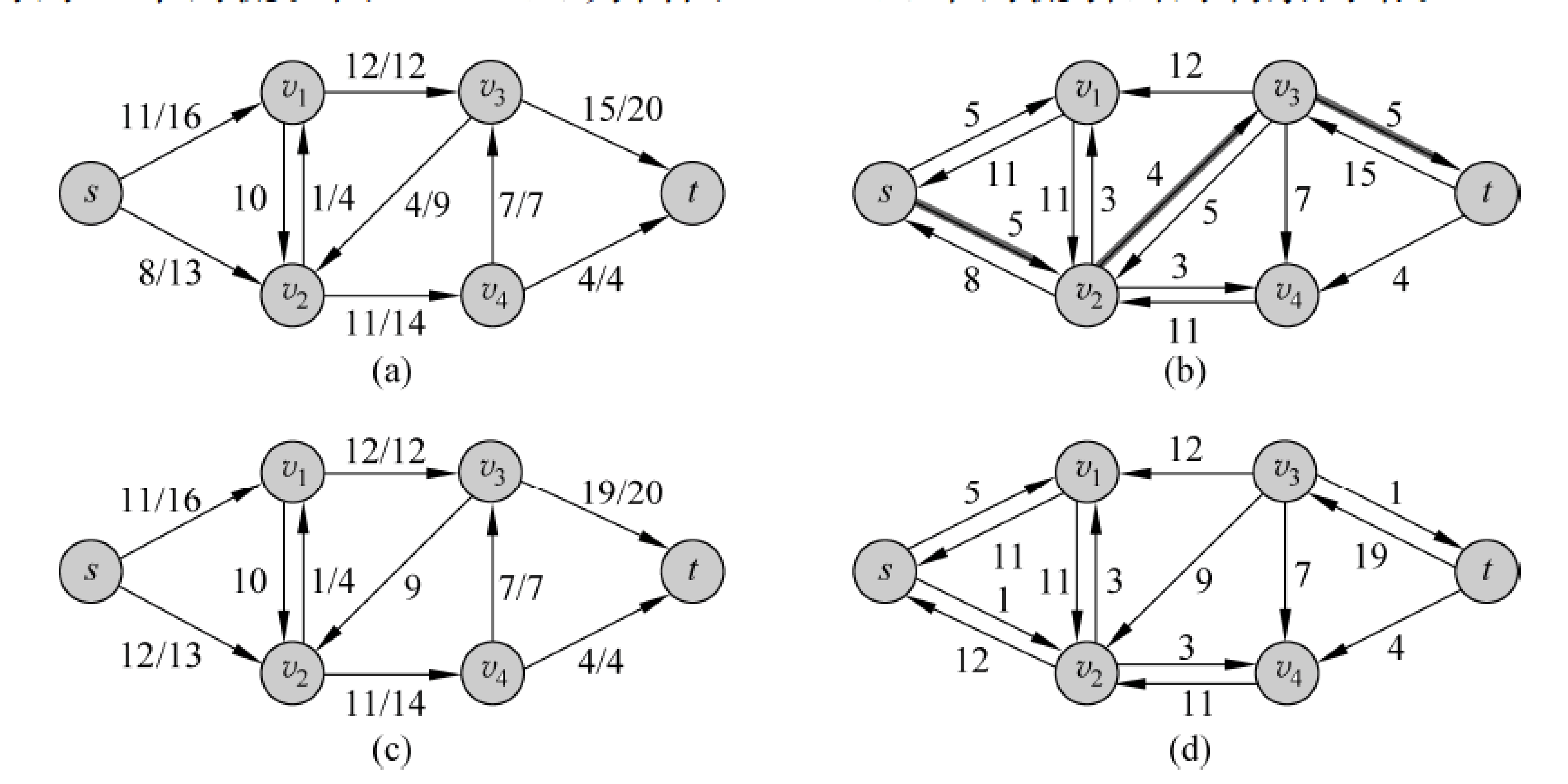


图 10-14　剩余网络与增广路径

设 $G=(V,E)$ 为一流网络,f 为 G 中的一个流,p 为 G_f 中的一条增广路径。设 f_p 在式(10-5)中定义。定义函数 f' : $V\times V \to \mathbf{R}$,其中 $f'=f+f_p$。则 f' 为 G 中值为 $|f'|=|f|+|f_p|>|f|$ 的一个流。这是因为根据式(10-4),对所有的 $u,v\in V$, $f_p(u,v) \leqslant$

$c_f(u,v)=c(u,v)-f(u,v)$。所以

$$\begin{aligned} f'(u,v) &= (f+f_p)(u,v) = f(u,v)+f_p(u,v) \\ &\leqslant f(u,v)+(c(u,v)-f(u,v)) \\ &= c(u,v) \end{aligned}$$

这说明 $f'=f+f_p$ 是 G 的一个流。由于 $|f'|=|f+f_p|=|f|+|f_p|$，而 $|f_p|=c_f(p)>0$，所以 $|f'|>|f|$。

此性质说明，可以利用流网络关于流 f 的剩余网络中的增广路径使流网络 G 的流增值，并且有定理 10-1。

定理 10-1　若 f 是 G 的一个最大流，则 G 关于 f 的剩余网络 G_f 中不存在增广路径。

这是因为，若 G_f 中还有一条增广路径 p，根据式(10-5)定义的 f_p 是 G_f 的一个流，所以流和 $f+f_p$ 是 G 中一个其值严格大于 $|f|$ 的流。这与 f 是一个最大流矛盾。

6）**流网络的割**

流网络 $G=(V,E)$ 的一个割 (S,T) 是 V 的一个分为 S 和 $T=V-S$，且 $s\in S$ 及 $t\in T$ 的一个划分。若 f 是一个流，则跨越割 (S,T) 的**净流**定义为 $\sum_{u\in S}\sum_{v\in T}f(u,v)$。割 (S,T) 的**容量**为 $\sum_{u\in S}\sum_{v\in T}c(u,v)$。一个网络的**最小割**是该网络的容量最小的割。

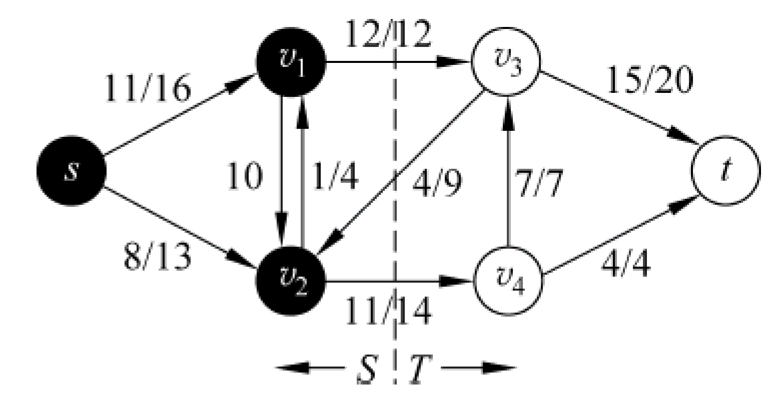


图 10-15　流网络的割

图 10-15 展示了图 10-13(b)中的流网络的割 $(\{s,v_1,v_2\},\{v_3,v_4,t\})$，其中 $S=\{s,v_1,v_2\}$ 而 $T=\{v_3,v_4,t\}$。S 中的顶点是黑色的，T 中的顶点是白色的。跨越 (S,T) 的净流是 $f(S,T)=19$，其容量为 $c(S,T)=26$。跨越该割的净流为

$$f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4)=12+(-4)+11=19$$

且其容量为

$$c(v_1,v_3)+c(v_2,v_4)=12+14=26$$

设 f 是流网络 G 中的一个流，源点为 s，汇点为 t，设 (S,T) 为 G 的一个割。则跨越 (S,T) 的净流 $\sum_{u\in S}\sum_{v\in T}f(u,v)=|f|$。这是因为

$$\begin{aligned} \sum_{u\in S}\sum_{v\in T}f(u,v) &= \sum_{u\in S}\sum_{v\in V-S}f(u,v) && \text{（根据割的定义）} \\ &= \sum_{u\in S}\sum_{v\in V}f(u,v)-\sum_{u\in S}\sum_{v\in S}f(u,v) && \text{（和式性质）} \\ &= \sum_{u\in S}\sum_{v\in V}f(u,v) && \text{（根据流的斜对称性）} \\ &= \sum_{v\in V}f(s,v)+\sum_{u\in S-s}\sum_{v\in V}f(u,v) && \text{（和式性质）} \\ &= \sum_{v\in V}f(s,v) && \text{（根据流的守恒性质）} \\ &= |f| && \text{（流的值定义）} \end{aligned}$$

引理 10-6　流网络 G 中的任一个流 f，以 G 的任一割的容量为上界。

这是因为根据上面的讨论有

$$|f| = \sum_{u\in S}\sum_{v\in T} f(u,v) \leqslant \sum_{u\in S}\sum_{v\in T} c(u,v) \qquad \text{(根据流的定义)}$$

等号在流 f 的值 $|f|$ 最大且割 (S,T) 的容量 $\sum_{u\in S}\sum_{v\in T} c(u,v)$ 最小时取得。

利用引理 10-6,下面来证明定理 10-2。

定理 10-2 设流网络 G 关于流 f 不存在增广路径,则 f 是 G 的一个最大流。

由于 G_f 中没有增广路径,即在 G_f 中没有从 s 到 t 的路径。定义

$$S=\{v\in V : G_f \text{ 中存在从 } s \text{ 到 } v \text{ 的路径}\}$$

及 $T=V-S$。划分 (S,T) 是一个割:显然 $s\in S$,而 $t\notin S$,这是因为在 G_f 中没有从 s 到 t 的路径。每一对满足 $u\in S$ 且 $v\in T$ 的顶点,有 $f(u,v)=c(u,v)$,这是因为若否,$(u,v)\in E_f$,这意味着 v 在 S 中。因此 $|f| = \sum_{u\in S}\sum_{v\in T} f(u,v) = \sum_{u\in S}\sum_{v\in T} c(u,v)$。根据引理 10-6 可知,$f$ 是 G 的一个最大流。

2. 算法的伪代码描述

定理 10-1 和定理 10-2 实际上告诉我们流网络 G 的一个流 f 是其一个最大流当且仅当 G 关于 f 的剩余网络 G_f 中不存在增广路径。这就引出了一个计算流网络最大流的算法:从流 f_0 开始,通过一系列的迭代,每次迭代寻求一条增广路径 p,并对 p 上的每条边的流加上剩余容量 $c_f(p)$ 得到新的更大的流 f,计算流网络关于 f 的剩余网络 G_f,直至 G_f 中不存在增广路径为止。这一思想用伪代码描述如下。

```
EDMONDS-KARP(G,s,t,c)
1 f←f0
2 cf←c
3 Gf←G
4 (π,d)←BFS(Gf,s)
5 while d[t]≠∞
6   do p←由 π 决定的从 s 到 t 的路径
7      cp←min{cf(u,v) : (u,v)在 p 中}
8      for p 中的每条(u,v)
9        do f[u,v]←f[u,v]+cp
10          f[v,u]←-f[u,v]
11          cf[u,v]←cf[u,v]-cp
12          cf[v,u]←cf[v,u]+cp
13     Gf←由 cf 决定的有向图
14     (π,d)←BFS(Gf,s)
15 return f
```

算法 10-15 解决最大流问题的算法 EDMONDS-KARP

第 1 行初始化流 f 为 f_0。第 2 行将容量矩阵 c_f 初始化为 c。第 3 行将剩余网络 G_f 初始化为 G。第 4 行调用广度优先搜索算法 BFS,对 G_f 计算以 s 为源顶点的广度优先树 π 和 s 到图中各顶点的最短距离 d。第 5~14 行的 **while** 循环反复寻求 G_f 中的一条增广路径

p（BFS(G_f, s)返回的数组 d 记录了 G_f 中从 s 到各顶点的最短距离，$d[t] \neq \infty$ 蕴含着 G_f 中存在增广路径），第 7 行计算剩余容量 c_p。通过第 8～12 行的 **for** 循环对增广路径 p 上每一边计算剩余流。第 13 行根据增广后的容量矩阵计算剩余网络 G_f。第 14 行调用广度优先搜索算法 BFS，对 G_f 计算以 s 为源顶点的广度优先树 π 和 s 到图中各顶点的最短距离 d。当不存在增广路径时，f 就是一个最大流。第 15 行将其返回。图 10-16 展示了 EDMONDS-KARP 在一个样本上运行的每一次迭代的结果。图 10-16(a)～图 10-16(d)是 **while** 循环的连续的迭代。每一部分的左边展示的是第 4 行确定的剩余网络 G_f，带阴影的是其增广路径 p。每一部分的右边展示了在 f 上加上 f_p 的结果。图 10-16(a)中的剩余网络就是输入网络 G。图 10-16(e)是 **while** 循环最后检测的剩余网络。其中已无增广路径了，所以展示在图 10-16(d)中的流 f 就是最大流。

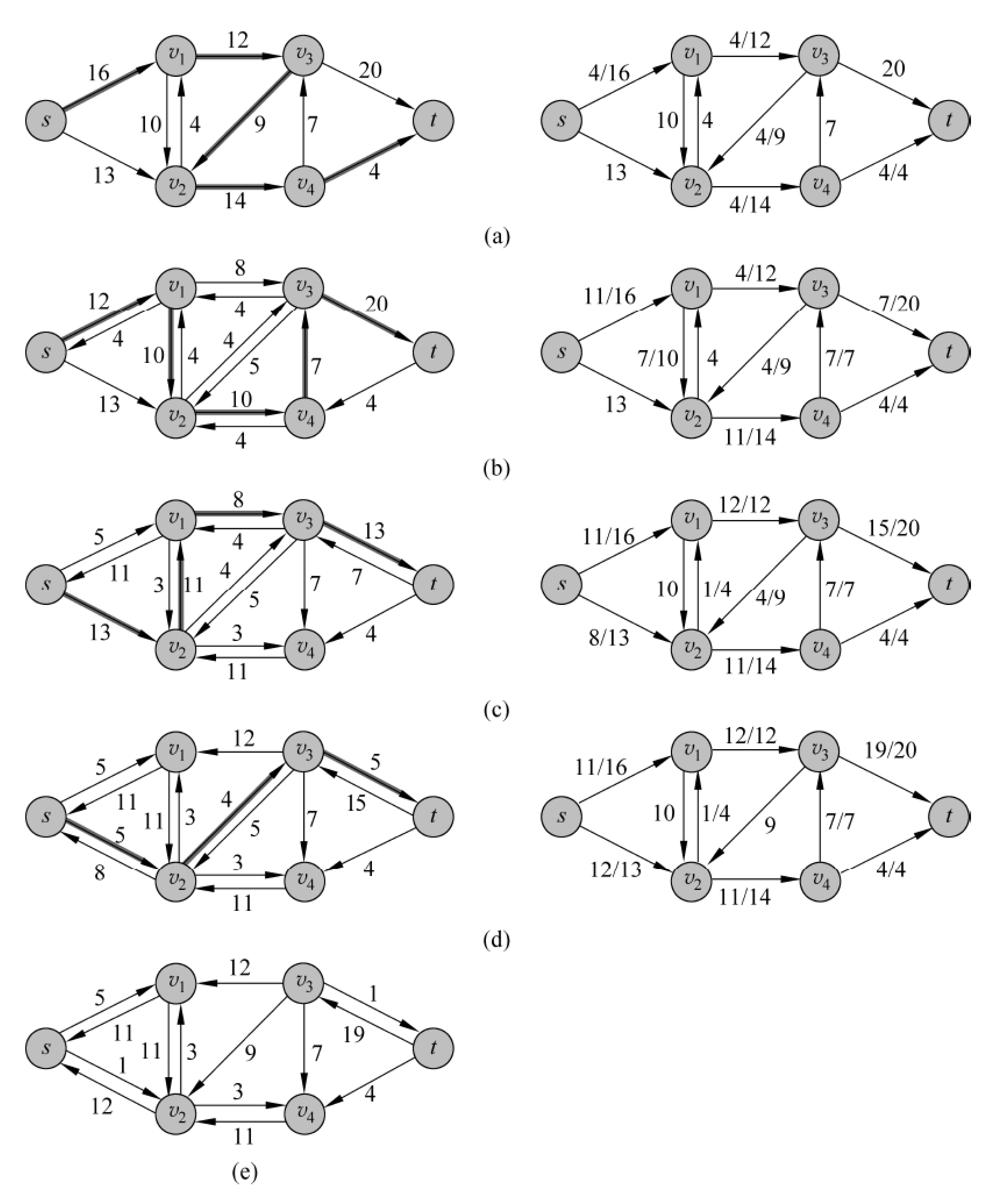


图 10-16　基本 Ford-KARP 算法的执行

3. 算法的运行时间

算法 EDMONDS-KARP 的运行时间取决于第 5～14 行的 **while** 循环的重复次数。为估算此循环的重复次数,要做些准备工作。若在剩余网络 G_f 中的增广路径 p 上边(u,v)的剩余容量为 p 的剩余容量,即若 $c_f[u,v]=c_p$,我们说(u,v)在增广路径 p 上是**临界**的。在沿着一条增广路径增加流之后,该路径上的任一临界边都从剩余网络中消失。此外,在任一条增广路径上,至少有一条临界边。下面来说明$|E|$条边中的每一条至多能成为$|V|/2-1$ 次临界边。为此,首先说明引理 10-7。

引理 10-7 算法 EDMONDS-KARP 运行于源点为 s、汇点为 t 的流网络 $G=(V,E)$,对任意 $v\in V-\{s,t\}$,其在剩余网络中的最短路径 $d_f[v]$随着流的增广而单调增加。

如果不是,假定对某顶点 $v\in V-\{s,t\}$,有一个流的增广使得从 s 到 v 的最短路径减少,我们将推出一个矛盾。设 f 是第一次造成某条最短路径减少的增广前的流,并设 f'恰为此增广后的流。设 v 为具有最小的 $d_{f'}[v]$的顶点,其距离由于该次增广而减少,所以 $d_{f'}[v]<d_f[v]$。设 $p=s\quad u\rightarrow v$ 是 $G_{f'}$ 中从 s 到 v 的一条最短路径,所以 $(u,v)\in E_{f'}$ 且

$$d_{f'}[u]=d_{f'}[v]-1 \tag{10-6}$$

根据对 v 的选取,我们知道到顶点 u 的距离并不减少,即

$$d_{f'}[u]\geqslant d_f[u] \tag{10-7}$$

我们断言$(u,v)\notin E_f$。这是因为,若有$(u,v)\in E_f$,则有

$$\begin{aligned} d_f[v] &\leqslant d_f[u]+1 && (\text{根据三角形不等式(见图 10-17)})\\ &\leqslant d_{f'}[u]+1 && (\text{根据不等式(10-7)})\\ &=d_{f'}[v] && (\text{根据等式(10-6)}) \end{aligned}$$

此与假设 $d_{f'}[v]<d_f[v]$矛盾。

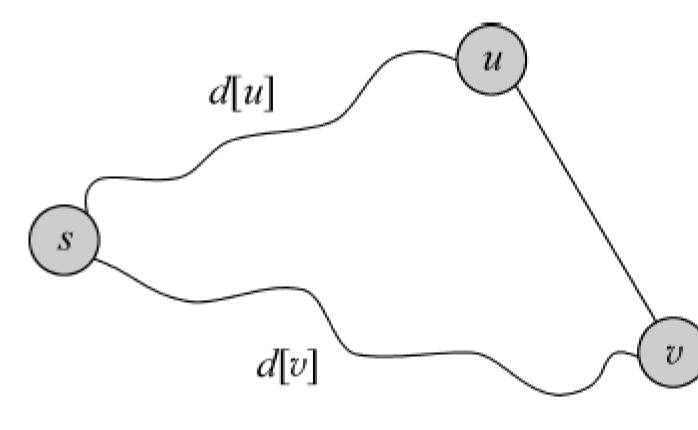


图 10-17 三个顶点间的路径

图 10-17 中,$(u,v)\in E$,从源点 s 到 u 的距离为 $d[u]$,到 v 的距离为 $d[v]$,u 到 v 的最短距离显然为 1,则最短路径 $s\quad u$、$s\quad v$ 和 $u\rightarrow v$ 之间必有称为三角形不等式的关系 $d[v]\leqslant d[u]+1$。

现在我们知道$(u,v)\notin E_f$ 且$(u,v)\in E_{f'}$。这意味着本次增广必使 v 到 u 的流有所增加。由于 EDMONDS-KARP 算法总是沿着最短路径增广流,所以 G_f 中从 s 到 u 的最短路径必以(v,u)作为最后一段边。于是

$$\begin{aligned} d_f[v] &=d_f[u]-1 && \\ &\leqslant d_{f'}[u]-1 && (\text{根据式(10-7)})\\ &=d_{f'}[v]-2 && (\text{根据式(10-6)}) \end{aligned}$$

此与 $d_{f'}[v]<d_f[v]$的假设矛盾。这样推出存在这样的顶点 v 是错误的。

利用引理 10-6,可以说明对每一条边$(u,v)\in E$,至多能成为$|V|/2-1$ 次临界边。事实上,由于增广路径是最短路径,当(u,v)首次为临界边时,有

$$d_f[v]=d_f[u]+1$$

一旦该流被增广,边(u,v)就在此剩余网络中消失。直至从 u 到 v 的流被减少,这只可能发生在(v,u)出现在一条增广路上。若 f'是该事件发生时 G 中的流,有

$$d_{f'}[u] = d_{f'}[v] + 1$$

根据引理 10-8，有 $d_f[v] \leqslant d_{f'}[v]$，有

$$d_{f'}[u] = d_{f'}[v] + 1 \geqslant d_f[v] + 1 = d_f[u] + 2$$

所以，从(u,v)变为临界到它下一次又变为临界为止，从源 s 到 u 的距离至少增加了 2。从源 s 到 u 的距离初始时至少为 0。从 s 到 u 的最短路径上的中间顶点不可能包含 s、u 或 t（这是因为(u,v)位于一增广界路径上意味着 $u \neq t$）。这样，即使直至 u 成为从源 s 不可达时，其距离至多为$|V|-2$。于是，(u,v)至多有$(|V|-2)/2=|V|/2-1$ 次可能成为临界的。由此可见，EDMONDS-KARP 过程的第 5～14 行的 **while** 循环的重复次数为 $O(EV)$。

定理 10-3　若 EDMONDS-KARP 算法运行于源为 s 汇为 t 的一个流网络 $G=(V,E)$上，则算法的运行时间为 $O(VE^2)$。

这是因为第 5～14 行的 **while** 循环的 $O(EV)$每一次重复执行时，第 8～12 行的 **for** 循环耗时 $O(V)$，第 14 行调用 BFS 耗时 $O(V+E)$（见 10.4 节对算法 BFS 的分析），由于所有顶点从 s 都是可达的，所以$|E| \geqslant |V|-1$，故每次重复耗时 $O(E)$。这样，就得到算法的运行时间为 $O(VE^2)$。

EDMONDS-KARP 算法还有一个很有趣的性质：若流网络 G 中的容量均取整数值，则 G 的最大流的值也是整数，这个特性称为**完备性**。完备性的正确性可以通过对如下事实的观察得到：EDMONDS-KARP 算法是从 f_0 开始，反复增广最终得到流网络 G 的最大流 f。每次增广得到的剩余网络中的容量都是整数值的增减，增加的流量也是整数值。

一些组合问题可以很容易地表示为最大流问题。下面介绍一个这样的问题：在一个二部图[①]中寻求最大匹配。为解决此问题，要利用 EDMONDS-KARP 算法所提供的完备性好处。

4. 二部图的最大匹配问题

给定一个无向图 $G=(V,E)$，一个**匹配**是边的一个子集 $M \subseteq E$ 使得对所有的 $v \in V$，至多有 M 中的一条边与 v 关联。若 M 中有一条与 v 关联的边，我们说一个顶点 v 是**匹配的**；否则，v 是**不匹配的**。一个**最大匹配**是含有最多边的一个匹配，即对于任一匹配 M'，有 $|M| \geqslant |M'|$。在本节中，将聚焦于在二部图中寻求最大匹配。假定顶点集合可以划分成$V=L \cup R$，其中 L 和 R 是不相交的且 E 中的所有边都连接 L 和 R 中的顶点。还假定 V 中的每一个顶点至少有一条边与之关联。图 10-18 示例了匹配的概念。

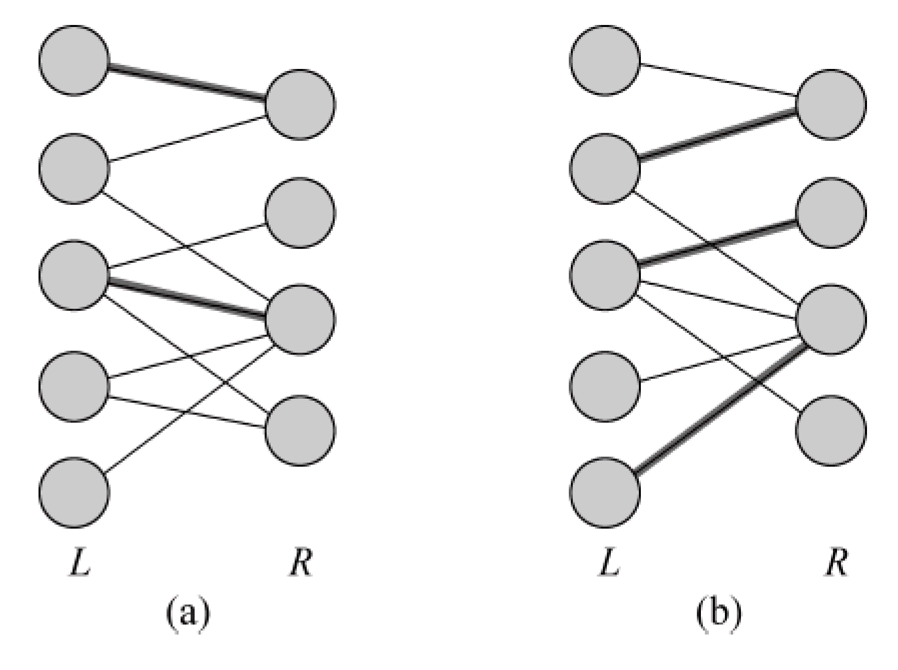


图 10-18　一个二部图 $G=(V,E)$，点划分为 $V=L \cup R$

图 10-18(a)为一个势为 2 的匹配。图 10-18(b)为一个最大匹配，其势为 3。

在二部图中寻求最大匹配问题有很多实际的应用。作为一个例子，可以考虑一个机器集合 L 和一个需要同时执行的任务集合 R。对一台具体的机器 $u \in L$ 及其能执行的一个具体任务 $v \in R$ 之间设置一条 E 中的边(u,v)。一个最大匹配使得尽可能多的机器得以运行。

① **二部图**是一个无向图 $G=(V,E)$，其中 V 可以分成两个集合 V_1 和 V_2，使得$(u,v) \in E$ 蕴涵着 $u \in V_1$ 且 $v \in V_2$ 或 $u \in V_2$ 且 $v \in V_1$，即所有的边都横跨集合 V_1 和 V_2。

5. 寻求二部图的最大匹配

可以利用 EDMONDS-KARP 算法在$|V|$和$|E|$的多项式时间内在一个无向二部图 $G=(V,E)$中寻求一个最大匹配。窍门在于构造一个流网络,其中的流对应于匹配,如图 10-19 所示。

图 10-19(a)由图 6-15 中的图 $G=(V,E)$得来的二部图,其顶点划分为 $V=L\cup R$。用带阴影的边表示出一个最大匹配。图 10-19(b)对应的流网络及其一个最大流。每一条边具有一个单位的容量。带有阴影的边具有流 1,而其他的边不带有流。从 L 到 R 的带有阴影的边对应于二部图中的最大匹配。对二部图 G 如下定义**对应流网络** $G'=(V',E')$。设源 s 和汇 t 不是 V 中顶点的新的顶点,并设 $V'=V\cup\{s,t\}$。若 G 的顶点划分为 $V=L\cup R$,G'的有向边是 E 中的边设置从 L 到 R 的方向连同 V 条的新边:

$$
\begin{aligned}
E' &=\{(s,u):u\in L\}\\
&\cup\{(u,v):u\in L,v\in R,且(u,v)\in E\}\\
&\cup\{(v,t):v\in R\}
\end{aligned}
$$

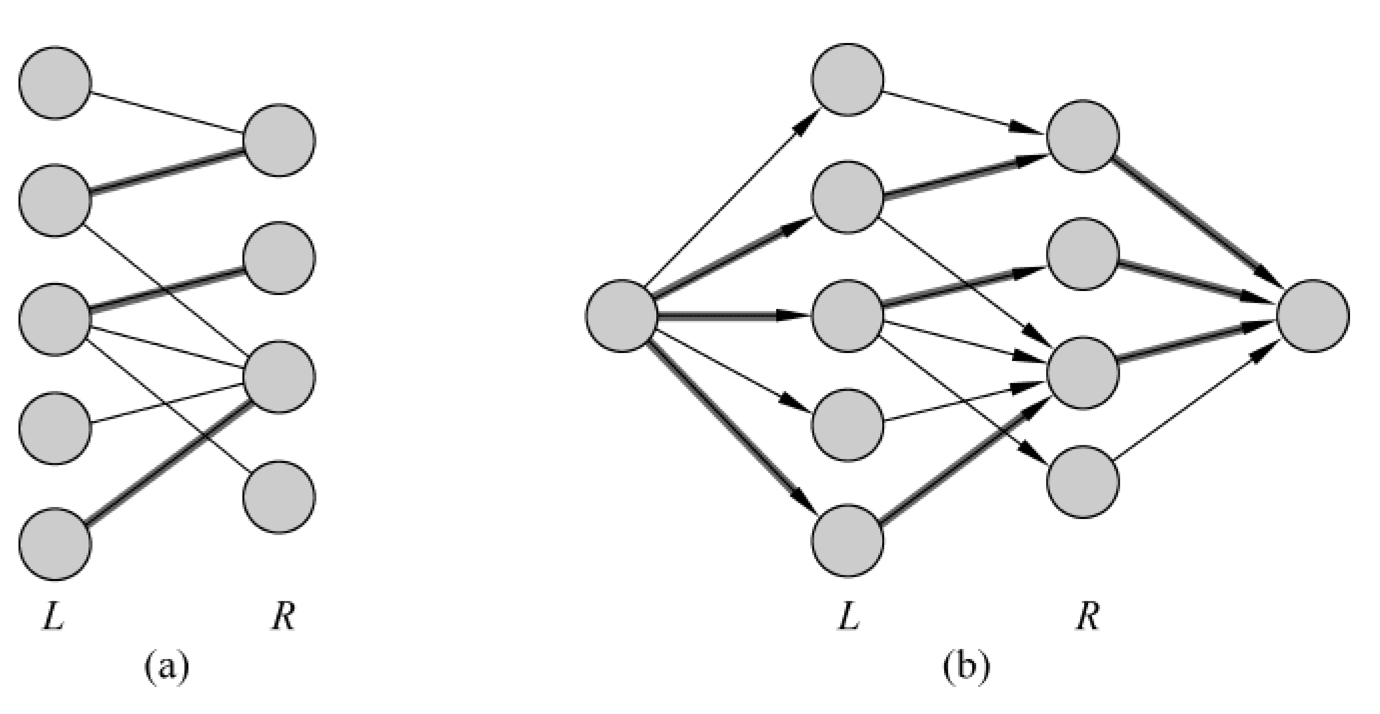


图 10-19 对应于一个二部图的流网络

为完成此构造,对 E'中的每一条边指派一个单位的容量。由于 V 中的每个顶点至少关联一条边,$|E|\geqslant|V|/2$。于是,$|E|\leqslant|E'|=|E|+|V|\leqslant3|E|$,所以$|E'|=\Theta(E)$。

引理 11-8 说明在 G 中的一个匹配直接对应于 G 的对应流网络 G中的一个流。若对所有的$(u,v)\in V\times V$,$f(u,v)$是一个整数,则称流网络 $G=(V,E)$中的一个流 f 是整数值的。

引理 10-8 设 $G=(V,E)$为一个二部图,其顶点划分为 $V=L\cup R$,并设 $G'=(V',E')$为其对应的流网络。若 M 是 G 的一个匹配,则在 G'中有一个整数值的流 f,使得$|f|=|M|$。反之,若f 是G'中的一个整数值流,则在 G 中有一个匹配 M 其所含边数$|M|=|f|$。

首先说明 G 中的一个匹配 M 对应于 G'中的一个整数值流。如下定义流 f:若$(u,v)\in M$,则$f(s,u)=f(u,v)=f(v,t)=1$ 以及 $f(u,s)=f(v,u)=f(t,v)=-1$。对所有其他的边$(u,v)\in E'$,定义 $f(u,v)=0$。不难验证 f 满足斜对称、容量限制及流守恒性质。

直观地看,每一条边$(u,v)\in M$ 对应于G'中通过路径 $s\to u\to v\to t$ 的流的一个单位。此外,由 M 中的各条边诱导出的各条路径除了 s 和 t 以外相互没有共同的顶点。跨越割$(L\cup$

$\{s\}, R \cup \{t\})$的净流等于$|M|$；于是，该流的值为$|f|=|M|$。

反之，设 f 为G'中的一个整数值流，并设 $M=\{(u,v): u\in L, v\in R$，且 $f(u,v)>0\}$。

对于每一个顶点 $u\in L$ 只有一条进入边，即(s,u)，且其容量为 1。于是，对于每一个 $u\in L$ 至多有一个单位正流进入它，并且若有一个单位的正流进入该顶点，根据流守恒性质必有一个单位的正流离开它。并且因为 f 是整数值的流，对每一个 $u\in L$，一个单位的流至多由一条边进入并且至多由一条边离开。于是，一个单位的正流进入 u 当且仅当恰有一个顶点 $v\in R$ 使得 $f(u,v)=1$，并且对每一个 $u\in L$，至多有一条带走正流的边离开它。对每一个 $v\in R$ 可以做出对称的讨论，所以本引理定义的集合 M 是一个匹配。

为领会$|M|=|f|$，注意对每一个得到匹配的顶点 $u\in L$，有 $f(s,u)=1$，并且对每一条边$(u,v)\in E-M$，有 $f(u,v)=0$。从而，

$$
\begin{aligned}
|M| &= \sum_{u\in L}\sum_{v\in R} f(u,v) = \sum_{u\in L}\sum_{v\in V'-L-\{s,t\}} f(u,v) \\
&= \sum_{u\in L}\Big(\sum_{v\in V'} f(u,v) - \sum_{v\in L} f(u,v) - f(u,s) - f(u,t)\Big) \\
&= \sum_{u\in L}\sum_{v\in V'} f(u,v) - \sum_{u\in L}\sum_{v\in L} f(u,v) - \sum_{u\in L} f(u,s) - \sum_{u\in L} f(u,t)
\end{aligned}
$$

流守恒性质蕴含着 $\sum_{u\in L}\sum_{v\in V'} f(u,v)=0$；流的斜对称性质蕴含着 $\sum_{u\in L}\sum_{v\in L} f(u,v)=0$ 和 $-\sum_{u\in L} f(u,s)=\sum_{u\in L} f(s,u)$；且由于没有边从 L 到 t，有 $\sum_{u\in L} f(u,t)=0$。于是，

$$
\begin{aligned}
|M| &= \sum_{u\in L} f(u,s) \\
&= \sum_{u\in V'} f(u,s) \qquad \text{（因为所有离开 } s \text{ 的边都进入 } L\text{）} \\
&= |f| \qquad \text{（根据 } |f| \text{ 的定义）}
\end{aligned}
$$

根据引理 10-8，可以推导出二部图 G 的一个最大匹配对应于其对应的流网络 G'中的一个最大流，并且可以因此通过在 G'上运行最大流算法 EDMONDS-KARP 而计算出 G 的一个最大匹配。

10.5.2　程序实现

很容易在 C 中实现算法 10-15 过程 EDMONDS-KARP。

```
double *edmondsKarp(double *c,int n,int s,int t){
    int u,v,*d,*pi;
    double *cf=(double*)malloc(n*n*sizeof(double)),
        *f=(double*)calloc(n*n,sizeof(double));
    Graph *gf;
    pair p;
    memcpy(cf,c,n*n*sizeof(double));
    gf=createGraph(cf,n);
    p=bfs(gf,s);
    pi=(int*)p.first;d=(int*)p.second;
    while(d[t]<INT_MAX){                    /*存在增广路径*/
```

```
12        double cp=DBL_MAX;
13        v=t;
14        u=pi[v];
15        while(u!=-1){                          /* 在增广路径扫描求出最小容量 */
16            if(cf[u*n+v]<cp)
17                cp=cf[u*n+v];
18            v=u;
19            u=pi[v];
20        }
21        for(v=t,u=pi[v];u!=-1;v=u,u=pi[v]){
22            cf[u*n+v]-=cp;
23            cf[v*n+u]+=cp;
24            f[u*n+v]+=cp;
25            f[v*n+u]-=cp;
26        }
27        free(pi);free(d);graphClear(gf);free(gf);
28        gf=createGraph(cf,n);
29        p=bfs(gf,s);pi=(int *)p.first;d=(int *)p.second;
30    }
31    free(cf);free(pi);free(d);graphClear(gf);free(gf);
32    return f;
33 }
```

程序 10-14　实现算法 10-15 过程 EDMONDS-KARP 的 C 源代码

对程序 10-14 的说明如下。

(1) 函数 edmondsKarp 实现算法 10-15 中的过程 EDMONDS-KARP。参数 c 表示流网络中各条边的容量构成的矩阵(按行优先原则存储在一维数组中)。参数 n 表示流网络中的顶点个数。参数 s、t 分别表示网络的源点和汇点。与算法一样,函数将计算并返回由容量矩阵 c 确定的流网络的一个最大流 f(表示成一维数组的矩阵)。

(2) 第 3 行声明的动态数组 cf 表示计算过程中剩余网络容量矩阵,在第 7 行初始化为原网络的容量矩阵 c。第 4 行声明的动态数组 f 表示计算过程中的网络的流,也是计算结果的最大流的存储空间,初始化为 0 矩阵(注意调用 calloc 函数为其分配空间)。第 5 行声明的 Graph 型指针变量 g 用来表示网络对应的图。因为在计算过程中需要通过对这个图进行广度优先搜索得到源点到汇点的增广路径。

(3) 第 11～30 行的 **while** 循环实现算法 10-15 中第 5～14 行的操作。其中,第 12～20 行对应算法中第 7 行计算增广路上的最小容量 cp。第 21～26 行的 **for** 循环对应算法中的第 8～12 行的操作,计算剩余网络的流 f。第 28 行和第 29 行对应算法中的第 14 行的操作,对剩余网络对应的图 g 做广度优先搜索。

为便于代码重用,把程序 10-14 存储在文件夹 graph 中的头文件 edmodskarp. h(函数原型声明)和源文件 edmondskarp. c(函数定义代码)中。

10.5.3　应用

1. 因特网带宽

Internet Bandwidth

On the Internet, machines (nodes) are richly interconnected, and many paths may exist between a given pair of nodes. The total message-carrying capacity (bandwidth) between two given nodes is the maximal amount of data per unit time that can be transmitted from one node to the other. Using a technique called packet switching, this data can be transmitted along several paths at the same time.

For example, the following figure (见图 10-20) shows a network with four nodes (shown as circles), with a total of five connections among them. Every connection is labeled with a bandwidth that represents its data-carrying capacity per unit time.

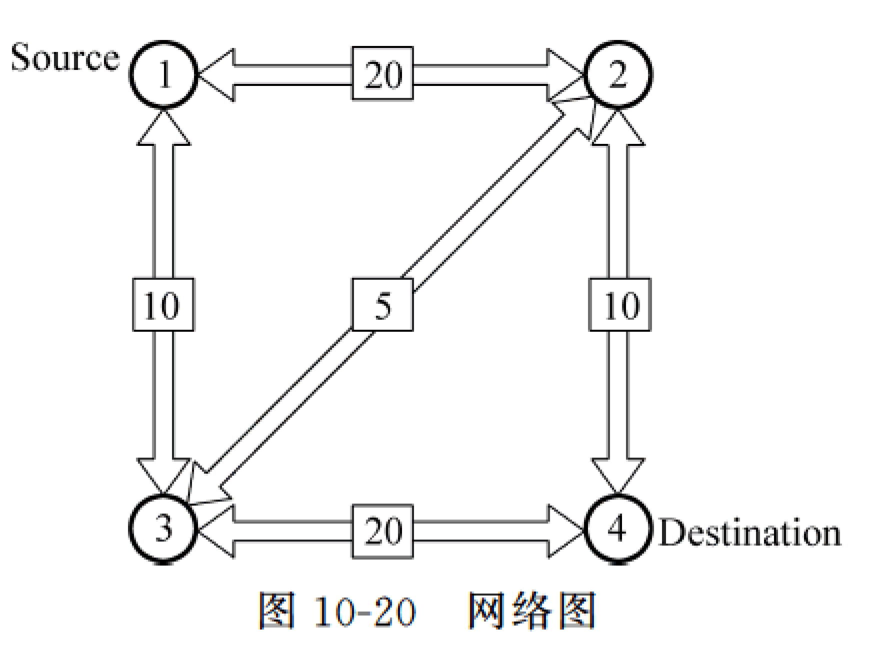


图 10-20　网络图

In our example, the bandwidth between node 1 and node 4 is 25, which might be thought of as the sum of the bandwidths 10 along the path 1-2-4, 10 along the path 1-3-4, and 5 along the path 1-2-3-4. No other combination of paths between nodes 1 and 4 provides a larger bandwidth.

You must write a program that computes the bandwidth between two given nodes in a network, given the individual bandwidths of all the connections in the network. In this problem, assume that the bandwidth of a connection is always the same in both directions (which is not necessarily true in the real world).

Input

The input file contains descriptions of several networks. Every description starts with a line containing a single integer n ($2 \leqslant n \leqslant 100$), which is the number of nodes in the network. The nodes are numbered from 1 to n. The next line contains three numbers s, t, and c. The numbers s and t are the source and destination nodes, and the number c is the total number of connections in the network. Following this are c lines describing the connections. Each of these lines contains three integers: the first two are the numbers of the connected nodes, and the third number is the bandwidth of the connection. The bandwidth is a non-negative number not greater than 1000.

There might be more than one connection between a pair of nodes, but a node cannot be connected to itself. All connections are bi-directional, i. e. data can be transmitted in both directions along a connection, but the sum of the amount of data transmitted in both directions must be less than the bandwidth.

A line containing the number 0 follows the last network description, and terminates the input.

Output

For each network description, first print the number of the network. Then print the total bandwidth between the source node *s* and the destination node *t*, following the format of the sample output. Print a blank line after each test case.

Sample Input	Output for the Sample Input
4 1 4 5 1 2 20 1 3 10 2 3 5 2 4 10 3 4 20 0	Network 1 The bandwidth is 25.

1）问题描述与分析

因特网上任意两台指定机器之间可能存在着多条路径。它们之间的“带宽”指的是这两个结点间在单位时间内流过的最大信息量。此问题中，一个输入案例给出了网络中的 n 个结点，若两个结点有直接连接，则给出连接带宽。要求计算源结点 s 和目标结点 t 之间的带宽。若将计算机结点视为流网络 G 中的顶点，任意两台计算机 u、v 间的连接视为两个顶点 u、v 间的双向边，连接的带宽视为边上的容量 $c[u,v]$，则问题归结为对给定的流网络 G 及指定的源顶点 s 和汇顶点 t 计算最大流的问题。需要注意的是，题中给出的一对顶点 u 和 v 间的带宽实际上给出了网络流中的两条边 (u,v) 和 (v,u)，它们具有相同的容量(带宽)。例如，图 10-20 中对应的网络流应当理解为图 10-21。

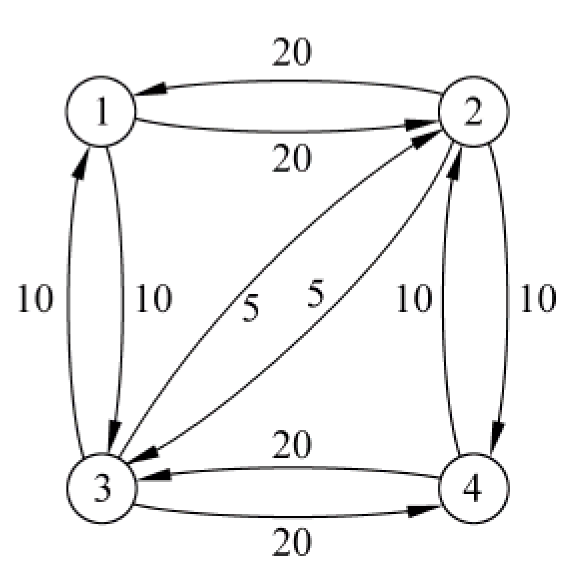


图 10-21　网络流图

对对应的流网络和源顶点和汇顶点调用算法 EDMONDS-KARP 过程就可解决 Internet Bandwidth 问题。

2）程序实现

```
int main(){
 int n,count=0;
 FILE *f1, *f2;
 assert(f1=fopen("chap10/Internet Bandwidth/inputdata.txt","r"));
 assert(f2=fopen("chap10/Internet Bandwidth/outputdata.txt","w"));
 fscanf(f1,"%d",&n);                              /*读取案例个数 n*/
 while(n){
     int s,t,m,i,u,v;
     double *c=(double*)calloc(n*n,sizeof(double)),w, *f;
     fscanf(f1,"%d%d%d",&s,&t,&m);
```

```
11      for(i=0;i<m;i++){
12          fscanf(f1,"%d%d%lf",&u,&v,&w);
13          c[(u-1)*n+(v-1)]=c[(v-1)*n+(u-1)]=w;
14      }
15      f=edmondsKarp(c,n,s-1,t-1);
16      for(w=i=0;i<n;i++)
17          w+=f[(s-1)*n+i];
18      fprintf(f2,"Network %d\n",++count);
19      fprintf(f2,"The bandwidth is %.0f\n",w);
20      free(c);free(f);
21      fscanf(f1,"%d",&n);
22    }
23    fclose(f1);fclose(f2);
24    return 0;
25 }
```

程序 10-15　解决 Internet Bandwidth 问题的 C 程序

对程序 10-15 的说明如下。

(1) 第 7～22 行的 **while** 循环处理输入文件中的每一个案例。其中,第 3 行声明的动态数组 c 用来表示本案例数据描述的流网络容量矩阵,初始化为 0 矩阵(用 calloc 函数分配空间)。

(2) 第 10～14 行从输入文件中读取本案例的数据填写网络的容量矩阵 c。第 15 行调用函数 edmondsKarp 计算本案例的最大流 f。

(3) 第 16 行和第 17 行计算流 f 的值 $|f|=\sum_{v\in V} f(s,v)$,第 18 行和第 19 行将计算结果写入输出文件。

程序 10-15 存储在文件夹 chap10/Internet Bandwidth 中的源文件 InternetBandwidth.c 中,读者可打开文件研读,并试运行。

2. 牛牛的美餐

Dining

Description

Cows are such finicky eaters. Each cow has a preference for certain foods and drinks, and she will consume no others.

Farmer John has cooked fabulous meals for his cows, but he forgot to check his menu against their preferences. Although he might not be able to stuff everybody, he wants to give a complete meal of both food and drink to as many cows as possible.

Farmer John has cooked F ($1\leqslant F\leqslant 100$) types of foods and prepared D ($1\leqslant D\leqslant 100$) types of drinks. Each of his N ($1\leqslant N\leqslant 100$) cows has decided whether she is willing to eat a particular food or drink a particular drink. Farmer John must assign a food type and a drink type to each cow to maximize the number of cows who get both.

Each dish or drink can only be consumed by one cow (i. e., once food type 2 is assigned to a cow, no other cow can be assigned food type 2).

Input

* Line 1: Three space-separated integers: N, F, and D.

* Lines 2..$N+1$: Each line i starts with two integers F_i and D_i, the number of dishes that cow i likes and the number of drinks that cow i likes. The next F_i integers denote the dishes that cow i will eat, and the D_i integers following that denote the drinks that cow i will drink.

Output

* Line 1: A single integer that is the maximum number of cows that can be fed both food and drink that conform to their wishes.

Sample Input

```
4 3 3
2 2 1 2 3 1
2 2 2 3 1 2
2 2 1 3 1 2
2 1 1 3 3
```

Sample Output

```
3
```

1) **问题描述与分析**

牛牛们都是些美食者。每个牛牛只钟情于某几种食物和饮料,而对其他的食物和饮料视而不见。若有 F 种食物,D 种饮料,把一种食物和一种饮料搭配分配给一个牛牛,问题是如何搭配能最大限度满足牛牛们的喜好。把 F 种食物视为二部图 G 中的顶点集 L,D 种饮料视为 G 中的顶点集 R。每个牛牛喜欢的食物和饮料可视为连接 L、R 的边。例如题中输入样例可表示为如图 10-22 所示的二部图。

图 10-22　二部图

这样,问题就转化为计算该二部图的最大匹配问题。

2) **程序实现**

```
int main(){
    int n,f,d,i;
    FILE *f1,*f2;
    double *fl,*c,max=0.0;
    assert(f1=fopen("chap10/Dining/inputdata.txt","r"));
    assert(f2=fopen("chap10/Dining/outputdata.txt","w"));
    fscanf(f1,"%d%d%d",&n,&f,&d);                    /*读取牛牛数 n、食物数 f、饮料数 d*/
    assert(c=(double*)calloc((f+d+2)*(f+d+2),sizeof(double)));
    for(i=1;i<=f;i++)                                /*添加一个源点*/
        c[i]=1.0;
     for(i=1;i<=d;i++)                               /*添加一个目标点*/
```

```
12      c[(f+i)*(f+d+2)+f+d+1]=1.0;
13    for(i=0;i<n;i++){                              /*根据二部图创建流网络*/
14      int fi,di,*u,*v,p,q;
15      fscanf(f1,"%d%d",&fi,&di);
16      assert(u=(int*)malloc(fi*sizeof(int)));
17      assert(v=(int*)malloc(fi*sizeof(int)));
18      for(p=0;p<fi;p++)
19        fscanf(f1,"%d",u+p);
20      for(q=0;q<di;q++)
21        fscanf(f1,"%d",v+q);
22      for(p=0;p<fi;p++)
23        for(q=0;q<di;q++)
24          c[u[p]*(f+d+2)+f+v[q]]=1.0;
25      free(u);free(v);
26    }
27    fl=edmondsKarp(c,f+d+2,0,f+d+1);               /*计算最大流*/
28    for(i=0;i<f+d+2;i++)                           /*计算最大匹配*/
29      max+=fl[i];
30    fprintf(f2,"%.0f\n",max);
31    free(c);free(fl);
32    fclose(f1);fclose(f2);
33    return 0;
34  }
```

程序 10-16　解决 Dining 问题的 C 程序

对程序 10-16 的说明如下。

（1）由于食品种数对应二部图中的顶点集 L，饮料种数对应二部图中的顶点集 R，所以二部图应该有 F+D 个顶点，添加一源点和一汇点，故对应于网络流的图 G 应有 F+D+2 个顶点。于是，程序中表示流网络容量矩阵的数组 c 要分配(f+d+2)*(f+d+2)个单元（第 8 行）。源点为 0，汇点为 f+d+1。第 9 行和第 10 行及第 11 行和第 12 行分别设置定源点到其他个顶点的容量为 1，其他各顶点到汇点的容量为 1。

（2）第 13～26 行读取输入文件中每个牛牛所喜欢的食品编号及饮料编号，用来设置网络中对应边上的容量（均为 1）。

（3）第 27 行调用函数 edmondsKarp 计算 c 对应的网络的最大流 f，第 28 行和第 29 行计算该流的值|f|，第 30 行将计算结果写入输出文件。

程序 10-16 存储在文件夹 chap10/Dining 中的源文件 dining.c 中，读者可打开文件研读，并试运行。

第 11 章　文本搜索

在信息处理中时常出现在一个文本中查找一个模式的发生位置的问题。例如，文本是正在编辑的一个文档，要查找的模式是用户提供的一个具体单词，如图 11-1 所示。这一问题称为**串匹配**问题。图 11-1 的目标是在文本 $T=\mathrm{ABCABAABCABAC}$ 中查找模式 $P=\mathrm{ABAA}$ 的首次发生。该模式在文本中仅出现了一次，偏移量为 $s=3$。模式的每一个字符通过一根竖线与文本中匹配的字符连接，所有匹配的字符显示有阴影。设所有合法字符构成的集合 Σ 是一个有限集，称为字母表。Σ^* 表示用字母表 Σ 中的字符构成的所有有限长度的串的集合。零长度**空串**用 ε 表示，也属于 Σ^*。人们将文本搜索中的文本和模式均视为 Σ^* 中的字符串。

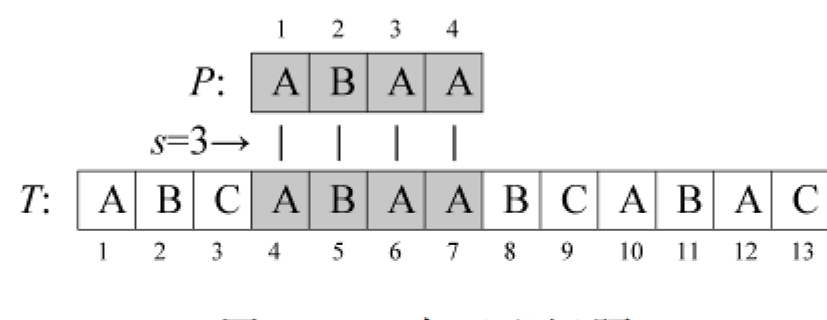


图 11-1　串匹配问题

11.1　固定模式的串匹配

固定模式是指组成模式的字符及模式的长度在搜索前就确定了。该问题形式化描述如下。

输入：Σ^* 中的文本字符串 $T[1..n]$和模式字符串 $P[1..m]$，$m\leqslant n$。

输出：若有 $0\leqslant s\leqslant n-m$ 且 $T[s+1..s+m]=P[1..m]$(即若对 $1\leqslant j\leqslant m$，$T[s+j]=P[j]$)，输出首个这样的**偏移量** s，否则输出-1。

11.1.1　强力算法

对文本 $T[1..n]$和模式 $P[1..m]$解决串匹配问题的强力算法利用一个循环对 $n-m+1$ 个可能的 s 值的每一个检测条件 $P[1..m]=T[s+1..s+m]$来查找首个有效偏移量，如图 11-2 所示。

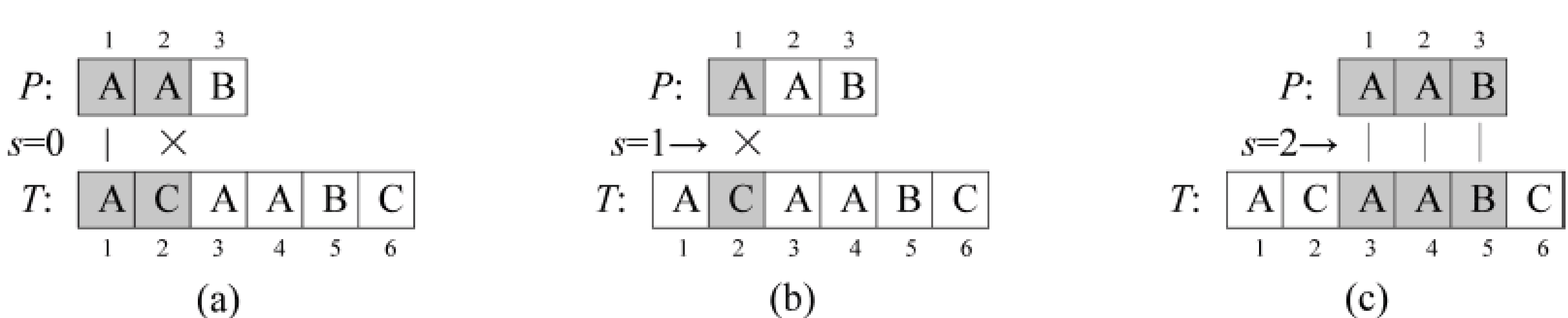


图 11-2　强力串匹配算法对模式 $P=\mathrm{AAB}$ 和文本 $T=\mathrm{ACAABC}$ 的操作

图 11-2 展示了对文本 T＝ACAABC 和模式 P＝AAB 运行解决串匹配问题用强力算法操作的情形。在图 11-2(a)中 $s=0$，$P[1]$与 $T[1]$匹配(用竖线相连)，但是 $P[2]\neq T[2]$，遇到一个失配(用一个叉表示)，s 自增 1 进入图 11-2(b)。在图 11-2(b)中 $s=1$，$P[1]\neq T[2]$，又遇到一个失配，s 自增 1，进入图 11-2(c)。在图 11-2(c)中 $s=2$，$P[1]=T[3]$，$P[2]=T[4]$，$P[3]=T[5]$，得到一个完整的匹配。

将此想法描述成伪代码过程如下。

```
NAIVE-STRING-MATCHER(T, P)
1 n←length[T]
2 m←length[P]
3 for s←0 to n−m
4   do k←1
5      while P[k]=T[s+k]
6         do k←k+1
7            if k>m
8               then return s
9 return −1
```

算法 11-1　计算文本为 T[1..n]、模式为 P[1..m]的串匹配的强力算法

NAIVE-STRING-MATCHER 过程的运行时间主要在于第 5 行的比较运算 $P(k)=T[s+k]$的执行次数，最坏情形是每次失配都发生在 $P[m]$与 $T[s+m]$处，即 s 从 0 到 $n-m$ 的 $n-m+1$ 个取值匹配都需做 m 次比较。因此运行时间是 $O((n-m+1)m)$(见图 11-3)。

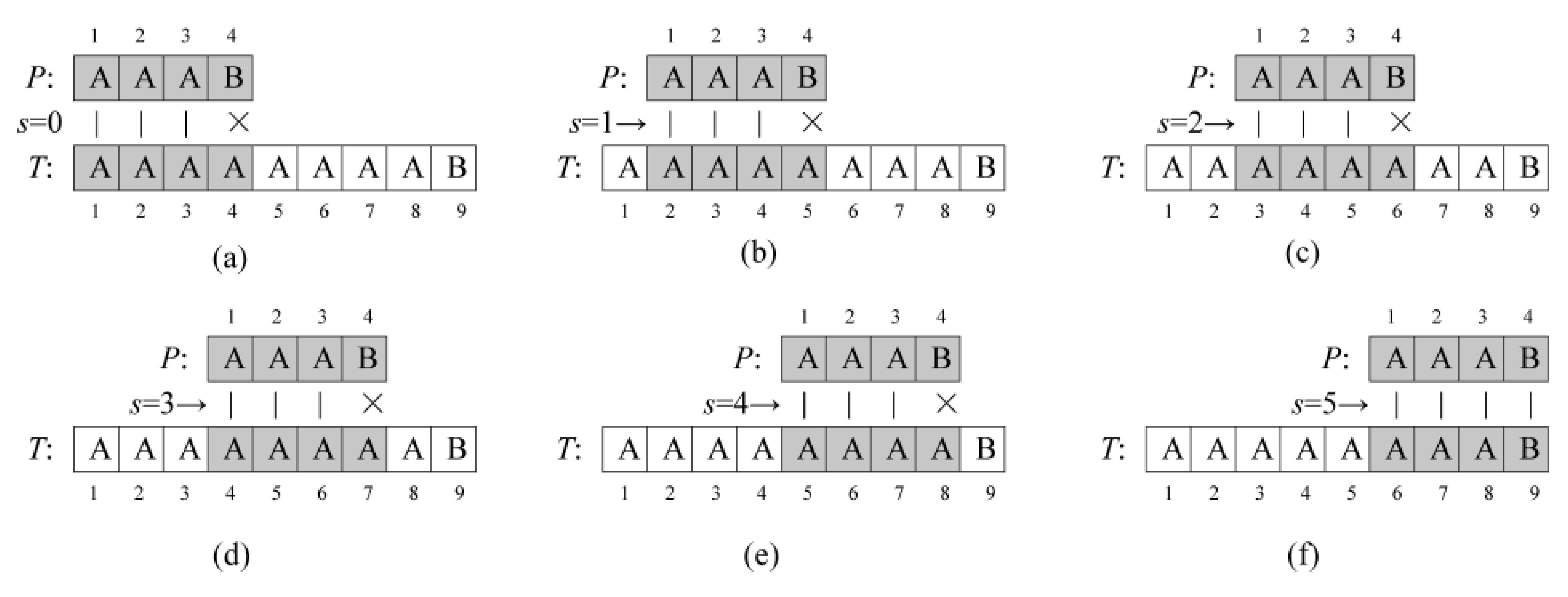


图 11-3　NAIVE-STRING-MATCHER 过程面对的一个最坏情形

图 11-3 展示了强力匹配算法对 $n=9$，$m=4$ 的一个最坏情形 T＝AAAAAAAAB，P＝AAAB 的执行过程：图 11-3(a)～图 11-3(f)部分中对 s 的每一种合法取值(共有 6 个值：0，1，2，3，4，5)，模式的每个元素(共有 4 个带有阴影的元素)都要与对应的文本元素(带有阴影的元素)进行比较，比较次数为 6×4＝24。事实上，就上例而言，不是每次匹配都要比较4 对元素。因为除了第一次匹配时比较了 4 对元素外，每得到一个失配(第一次出现在 $s=0$，$i=4$ 处，最后一次出现在 $s=4$，$i=4$ 处)，我们观察到，模式中的 $P[1..3]$是与文本中的 $T[s+1..s+3]$匹配的，并且$P[1..3]$的前缀 $P[1..2]$恰为 $T[s+1..s+3]$的后缀。这样，当将偏移量 s 自增 1 后，就知道 $T[s+1..s+2]$与 $P[1..2]$是匹配的，所以比较可以从

$i=3$ 开始进行。这样,就只需比较两对元素($P[3]$、$T[s+3]$和 $P[4]$、$T[s+4]$),大大减少了比较的次数,如图 11-4 所示。

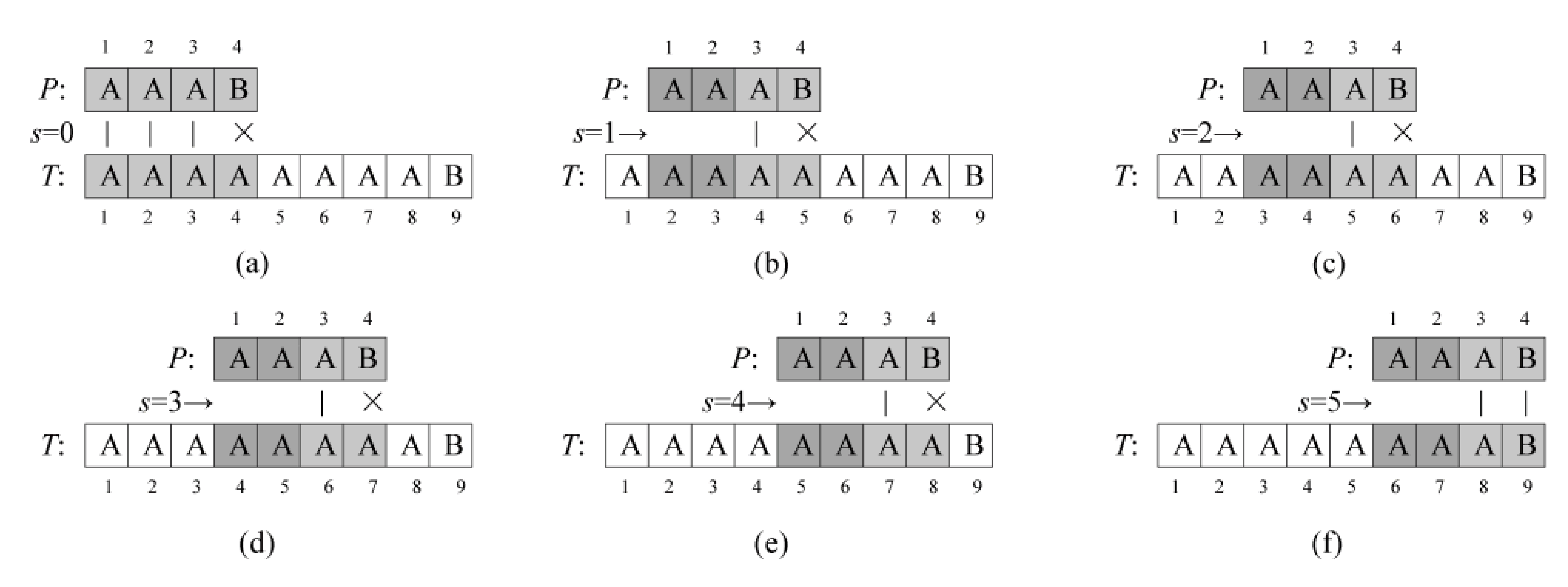


图 11-4 利用模式自身结构提供的信息改善模式匹配的运行时间效率

图 11-4 展示了对文本 T=AAAAAAAAB,模式 P=AAAB 的匹配在知道了模式自身的结构特点 $P[1..2]$=AA=$P[2..3]$,即 $P[1..3]$=AAA 的前缀 $P[1..2]$=AA 恰为 $P[1..3]$=AAA 的后缀 $P[2..3]$的前提下,每次 s 移动后,匹配过程中的比较次数可大大减少。图 11-4(a)中模式所有的 4 个元素均与对应文本元素比较,图 11-4(b)~图 11-4(f)部分,每次仅对模式的最后两个元素分别与对应的文本元素进行比较(浅色阴影部分)。比较次数为$4+2\times5=14$。

11.1.2 KMP 算法

由此可见,利用模式 $P[1..m]$本身结构的信息,对于 $2\leqslant i\leqslant m$,$P[1..i-1]$的前缀 $P[1..k]$ $(k<i-1)$若同时亦为 $P[1..i-1]$的后缀 $P[i-k..i-1]$,即 $P[1..k]=P[i-k..i-1]$可以提高匹配的速度。设 $\pi[i]$为 $P[1..i-1]$中相等前后缀的长度最大者,即

$$\pi[i]=\max\{k\mid 0\leqslant k<i-1\wedge P[1..k]=P[i-k..i-1]\} \tag{11-1}$$

$\pi[i]$在位置 i 处遇到失配时,确定新一轮匹配的偏移量和比较起始位置很有用。由于失配可能发生在 $1\sim m$ 内的任何位置,我们需要计算 $\pi[1..m]$。而 $i=1$ 的情形并不包含在式(11-1)中,约定 $\pi[1]=0$。此后,将 $\pi[1..m]$称为模式 $P[1..m]$的**失配函数**。

下面,以模式 $P[1..7]$=AABAABC 为例,考察其失配函数 $\pi[1..7]$。

$i=1$ 时,按上述约定 $\pi[1]=0$。

$i=2$ 时,$P[1..i-1]$=A,前缀为{ε},后缀为{ε},前、后缀最长公共串为 ε,其长度 $k=0$,$\pi[2]=k=0$。

$i=3$ 时,$P[1..i-1]$=AA,前缀为{A},后缀为{A},前后缀最长公共串为 A,其长度 $k=1$,$\pi[3]=k=1$。

$i=4$ 时,$P[1..i-1]$=AAB,前缀为{A, AA},后缀为{AB, B},前、后缀最长公共串为 ε,其长度 $k=0$,$\pi[4]=k=0$。

$i=5$ 时,$P[1..i-1]$=AABA,前缀为{A, AA, AAB},后缀为{ABA, BA, A},前、后缀最长公共串为 A,其长度 $k=1$,$\pi[5]=k=1$。

$i=6$ 时，$P[1..i-1]=$ AABAA，前缀为{A，AA，AAB，AABA}，后缀为{ABAA，BAA，AA，A}，前、后缀最长公共串为 AA，其长度 $k=2$，$\pi[6]=k=2$。

$i=7$ 时。$P[1..i-1]=$ AABAAB，前缀为{A，AA，AAB，AABA，AABAA}，后缀为{ABAAB，BAAB，AAB，AB，B}，前、后缀最长公共串为 AAB，其长度 $k=3$，$\pi[7]=k=3$。

对任意模式 $P[1..m]$的失配函数 $\pi[1..m]$的计算方法稍后详细讨论，此处假定算法过程为 GET-PI(P)，它计算并返回模式 P 的失配函数 π。匹配过程中若在 i 处得到一个失配，函数值 $\pi[i]$在准备下一轮匹配的偏移量 s 和比较开始位置 i 的作用如图 11-5 所示。

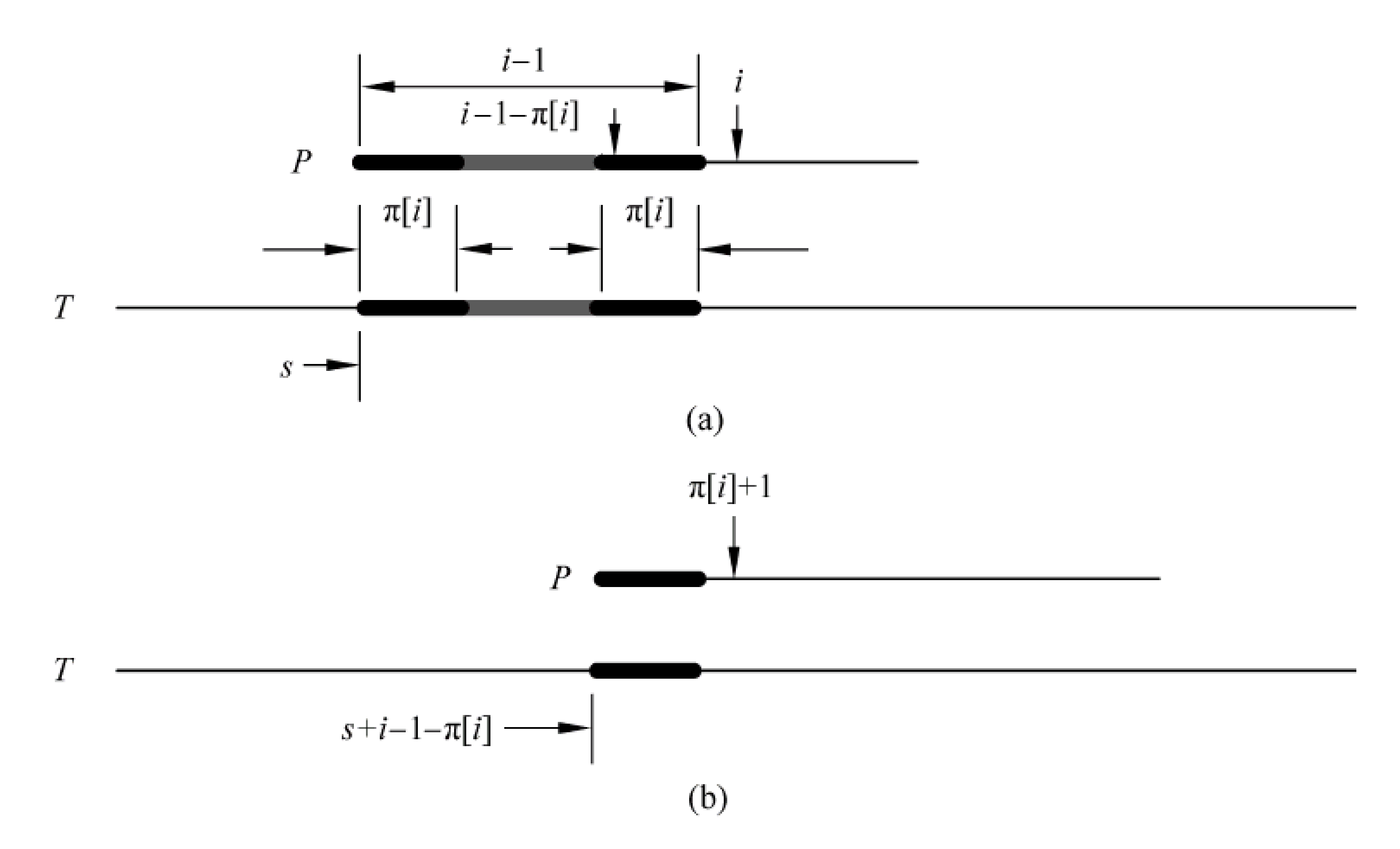


图 11-5　在位置 i 失配后确定新一轮匹配的偏移量 s 和比较起始位置 i 时 $\pi[i]$的意义

图 11-5 展示了在位置 i 处遇到失配后，如何用由式(11-1)定义的 $\pi[i]$确定下一轮匹配的偏移量 s 和比较开始位置 i。图 11-5(a)表示在一轮匹配过程中，$P[1..i-1]=T[s+1..s+i-1]$，但 $P[i]\neq T[s+i]$。$P[1..i-1]$在图中表示成粗线。根据 $\pi[i]$的定义，$P[1..i-1]$可分成三部分：$P[1..\pi[i]]$、$P[\pi[i]+1..i-\pi[i]-1]$和 $P[i-\pi[i]..i-1]$。其中前后两部分相等，表示成黑色的，中间部分表示成灰色的。在文本 T 中，由于 $T[s+1..s+i-1]=P[1..i-1]$，对应部分也表示成同样的形式。图 11-5(b)表示下一轮新的匹配开始时，偏移量可调整到 $s+i-1-\pi[i]$处。这是因为 $\pi[i]$记录的是 $P[1..i-1]$中相等前后缀最长者的长度，所以 $T[s+1..s+i-1](=P[1..i-1])$中不存在与 $P[1..m]$的任何长于 $\pi[i]$的前缀匹配的可能。对新的偏移量 s，$T[s+1..s+\pi[i]]=P[1..\pi[i]]$。这是因为这对应着图 11-5(a)中文本的第二根粗黑线段，它和同样表示成粗黑线段的 $P[1..\pi[i]]$相同。所以新的匹配过程中可以不再比较两者的对应字符，而从 $\pi[i]+1$ 开始，即 i 调整为 $\pi[i]+1$。

需要说明的是有一个例外，当失配发生在 $i=1$ 处，即 $T[s+1]\neq P[1]$。这时，由于我们人为地记 $\pi[1]$为 0，所以这一情形不能运用图 11-5 的所示方式确定下一轮匹配的偏移量 s 和比较开始位置 i。幸运的是，这种情况意味着下一轮匹配应该从文本的下一个字符开始与模式的从第一个字符开始依次比较，即让偏移量 s 自增 1，而 i 保持为 1。

将上述利用模式的失配函数改善匹配过程的思想写成如下伪代码。

```
KMP(T, P, π)
n←|T|, m←|P|
```

```
2 π←GET-PI(P)
3 s←0, i←1
4 while s+i≤n
5     do if T[s+i]=P[i]
6           then i←i+1
7                if i>m
8                   then return s
9           else if i=1
10                  then s←s+i
11                  else s←s+(i−1)−π[i]
12                       i←π[i]+1
13 return −1
```

算法 11-2　改进的串匹配算法

设文本 $T[1..23]$＝ABC ABCDAB ABCDABCDABDB，模式 $P[1..7]$＝ABCDABD。模式 P 的失配函数 $\pi[1..7]=\{0,0,0,0,0,1,2\}$。图 11-6 展示了运行 KMP($T$, P, π)的过程。

图 11-6(a)中，初始时 $s=0$，从 $i=1$ 开始依次比较模式与文本的对应字符 $T[s+i]$和 $P[i]$。在$i=4$ 处 $T[s+i]=T[4]=$“ ”$\neq$“D”$=P[4]=P[i]$，遇到一个失配。由于 $\pi[i]=\pi[4]=0$，所以将 s 移到 $s+(i-1)-\pi[i]=0+3-0=3$ 处，重置 $i=\pi[i]+1=1$。

图 11-6(b)中，此时 $s=3$，从 $i=1$ 开始，继续依次比较。马上遇到失配：$T[s+i]=T[4]=$“ ” $\neq$“A”$=P[1]=P[i]$。由于 $i=1$，这意味着需要立即将 s 推进一个位置，移到 $s+1=4$ 处，保持 $i=1$。

图 11-6(c)中，此时 $s=4$，从 $i=1$ 开始继续依次比较。在 $i=7$ 处，$T[s+i]=T[11]=$“ ”$\neq$“D”$=P[7]=P[i]$，遇到一个失配。由于 $\pi[i]=\pi[7]=2$，s 移到 $s+(i-1)-\pi[i]=8$，重置 i 为 $\pi[i]+1=3$。

图 11-6(d)中，此时 $s=8$，继续从 $i=3$ 开始依次比较。马上遇到一个失配：$T[11]=$“ ”$\neq$“C”$=P[3]$。由于 $i>1$。故设 s 为 $s+(i-1)+\pi[i]=10$，$i=\pi[i]+1=1$。

图 11-6(e)中，此时 $s=10$，$i=1$，继续依次比较。在 $i=1$ 处，$T[s+i]=T[11]=$“ ” $\neq$“A”$=P[1]=P[i]$，遇到一个失配。由于 $i=1$ 与图 11-6(b)中的情形一样。将 s 移到 $s+1=11$ 处，保持 $i=1$。

图 11-6(f)中，此时 $s=11$，从 $i=1$ 开始继续依次比较。在 $i=7$ 处，$T[18]=$“C”$\neq$“D”$=P[7]$遇到失配。$\pi[i]=\pi[7]=2$，将 s 移到 $s+(i-1)-\pi[i]=15$，重置 $i=\pi[i]+1=3$。

图 11-6(g)中，$s=15$ 时，从 $i=3$ 开始继续依次比较。这次得到一个完整的匹配。于是文本 T 中模式 P 首次出现在偏移量 $s=15$ 处。

整个过程的最基本操作就是第 5 行，即在偏移量为 s 时，对满足 $1\leq i\leq m$ 且 $s+i\leq n$ 的当前 i，比较 $T[s+i]$与 $P[i]$。根据比较的结果下次比较操作前，会有 3 种情形之一发生。

(1) $T[s+i]=P[i]$。除了图 11-6(b)、图 11-6(d)和图 11-6(e)一开始就遇到失配外，其他的图 11-6(a)、图 11-6(c)和图 11-6(f)在遇到失配前以及整个图 11-6(g)都处于此种情形。此时只需要让 i 自增 1(第 6 行)，并检测 i 是否大于 m。若是，意味着找到了一个完整匹配，返回偏移量 s(第 7 行和第 8 行)。

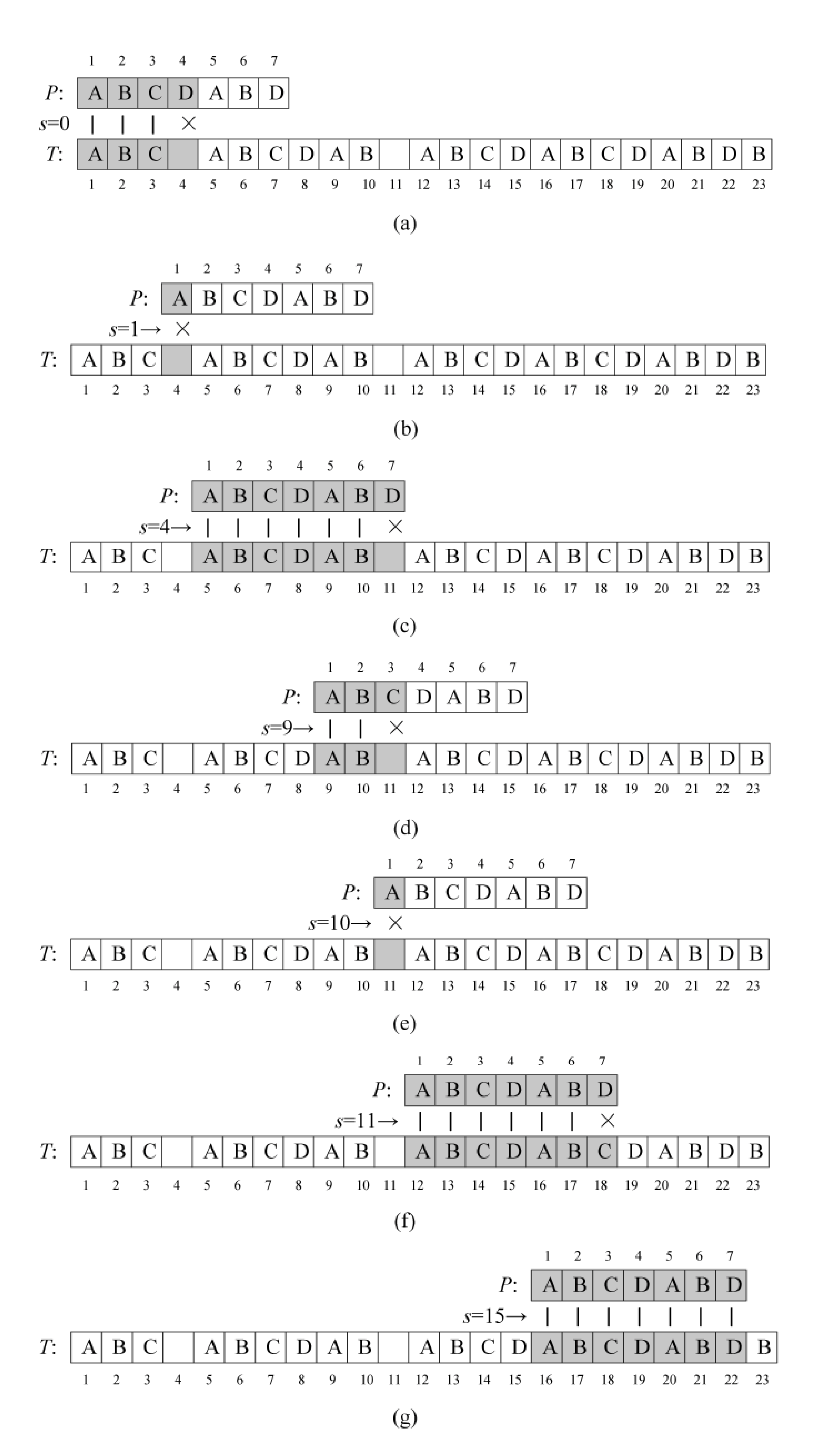


图 11-6　运行 KMP(*T*, *P*,π)的过程

(2) $T[s+i] \neq P[i]$,且 $i=1$。图 11-6(b)和图 11-6(e)就是这种情形。这意味着一轮匹配开始就失败了。这时要做的就是 s 自增 1,且 i 保持为 1(第 9 行和第 10 行)。

(3) $T[s+i] \neq P[i]$,且 $i \neq 1$。这意味着本轮匹配在 $i>1$ 处失配,由于 $\pi[i]$ 表示 $P[1..i-1]$的前缀 $P[1..\pi[i]]$与 $P[i-\pi[i]..i-1]$是相同的,所以可以将 s 直接推进$(i-1)-\pi[i]$个位置,且 i 可从 $\pi[i]+1$ 开始进行下一轮匹配(第 11 行和第 12 行)。

在匹配过程中,文本 $T[1..n]$中只有发生 $T[s+i]\neq P[i]$且 $i>1$ 时的文本中元素 $T[s+i]$才可能进行多于 1 次的比较,如图 11-6(a)、图 11-6(c)和图 11-6(d)中的情形。这是因为在此情形下,第 11 行和第 12 行的操作将 s 和 i 分别调整为 $s+(i-1)-\pi[i]$和 $\pi[i]+1$。s 由所长而 i 有所减,两者之和却不变(还是 $s+i$)。其他所有的元素仅做一次比较。上例中的 $T[4]$、$T[11]$就是这样的元素,其中 $T[4]$比较了 2 次,$T[11]$比较了 3 次。假定 $T[t]$是这样的元素,首次比较操作时,偏移量为 s,对应模式中的元素为 $P[i]$。第 2 次比较时,s 有所增长,但必小于 t。如果有第 3 次比较,也是如此。因此,重复比较次数至多为 $t-s$(首次比较时的值)。由 t 和 s 的单调增长性可知,所有这样的元素重复比较次数之和必不会超过 n。加上至多 n 个仅比较一次的元素,算法 KMP 中第 4～12 行 while 循环的重复次数至多为 $2n$ 次。

接下来,我们来考虑对一般的模式 $P[1..m]$计算其失配函数 $\pi[1..m]$的过程 GET-PI(P)。显然 $\pi[1]=\pi[2]=0$。对于 $i>2$,设已知 $\pi[1..i-1]$且 $\pi[i-1]=k$,即 $P[1..i-2]$的前、后缀中最长公共串是 $P[1..k]$,即 $P[1..k]=P[i-k-1..i-2]$,要计算 $\pi[i]$。

(1) 此时若有 $P[k+1]=P[i-1]$,则必有 $P[1..k+1]=P[i-k-1..i-1]$,且不会有 $k'>k+1$使得 $P[1..k']=P[i-k'..i-1]$。因此可得到 $\pi[i]=k+1=\pi[i-1]+1$。

(2) 若 $P[k+1]\neq P[i-1]$,又要分两种情况:其一,$k=0$,此时,必有 $\pi[i]=0$;其二,$k>0$,如图 11-7 所示。

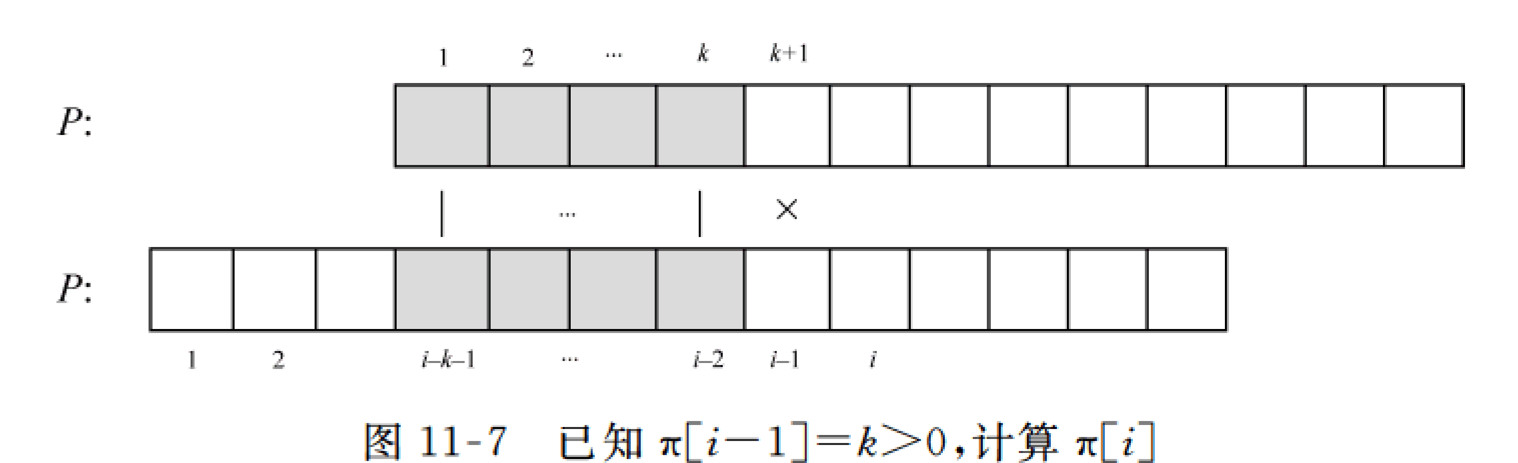


图 11-7　已知 $\pi[i-1]=k>0$,计算 $\pi[i]$

这意味着 $P[1..i-1]$的前、后缀公共串最大长度应当重新计算。从图 11-7 中可见,$P[1..i-1]$的前后缀的最长公共串除了最后一个元素为 $P[i-1]$以外,其他部分应该是 $P[1..k]$(灰色阴影部分元素)的前后缀的公共串。所以,k 的值可以从 $\pi[k]$起,递减地检测算得。

将这些思考写成如下的对模式 $P[1..m]$计算失配函数 $\pi[1..m]$的过程。

```
GET-PI(P)
m←length[P]
π[1]←π[2] ←0
i←3, k←π[i-1]
while i≤m
    do if P[i-1]=P[k+1]
        then i←i+1
            π[i]←k+1
            k←k+1
        else if k>0
            then k←π[k]
```

```
11                else π[i]←0
12                      i←i+1
13 return π
```

算法 11-3　计算模式 $P[1..m]$的失配函数 $\pi[1..m]$的算法

算法 11-3 的运行时间由第 4～12 行的 **while** 循环的重复次数决定。该循环的循环条件决定了至少重复 $m-1$ 次(若第 5 行检测的条件成立或 $P[k+1]\neq P[i-1]$且 $k=0$),每次这样的重复必执行 i 的自增 1 操作。换句话说,过程要执行 $m-1$ 次 k 的自增 1 操作。在这 $m-1$ 次伴随着 i 自增 1 的操作,第 8 行 k 也执行了自增 1 的操作。在整个循环的重复过程中第 10 行要执行 k 的减小操作(因为 $\pi[k]<k$)。然而,无论何时,k 非负。所以第 10 行的操作次数必不会超过 $m-1$。因此,算法 GET-PI(P)的运行时间为 $\Theta(m)$。回到算法 KMP(T, P),我们知道其中第 4～12 行 **while** 循环耗时为 $\Theta(n)$,加上现在知道其中第 2 行执行 GET-PI(P)耗时 $\Theta(m)$,所以 KMP 的运行时间为 $\Theta(n+m)$。由于必有模式的长度不超过文本的长度,即 $m\leqslant n$,故 KMP 算法的运行时间可表示为 $\Theta(n)$。

顺便说明,上述算法之所以称为 KMP 算法,是因为该算法是由是由 Knuth、Pratt 和 Morris 各自独立发明的。

11.1.3　程序实现

C 语言为程序员提供了大量的字符串处理库函数,包括常用的计算串长度函数、串连接函数、取子串函数、串比较函数等。同时还提供字符串匹配函数 strstr,该函数的用户接口如下:

```
char * strstr(char * str1, char * str2);
```

函数 strstr 表示在第一个参数表示的串 str1 中查找第二个参数表示的串 str2 第一次出现的位置指针(与本节中定义的偏移量 s 稍有差别),并以该指针作为返回值。

我们实现了 KMP 算法,C 代码写在源文件 textmatch.c 中,该文件存储于 utility 文件夹中,读者可打开该文件研读。为节省篇幅此处就不再详细列出代码了。

11.1.4　应用

DNA Laboratory

Description

Background

Having started to build his own DNA lab just recently, the evil doctor Frankenstein is not quite up to date yet. He wants to extract his DNA, enhance it somewhat and clone himself. He has already figured out how to extract DNA from some of his blood cells, but unfortunately reading off the DNA sequence means breaking the DNA into a number of short pieces and analyzing those first. Frankenstein has not quite understood how to put the pieces together to recover the original sequence.

His pragmatic approach to the problem is to sneak into university and to kidnap a number of smart looking students. Not surprisingly, you are one of them, so you would better come up with a solution pretty fast.

Problem

You are given a list of strings over the alphabet A (for adenine), C (cytosine), G (guanine), and T (thymine), and your task is to find the shortest string (which is typically not listed) that contains all given strings as substrings.

If there are several such strings of shortest length, find the smallest in alphabetical/lexicographical order.

Input

The first line contains the number of scenarios.

For each scenario, the first line contains the number n of strings with 1≤n≤15. Then these strings with 1≤ length≤100 follow, one on each line, and they consist of the letters "A","C", "G", and "T" only.

Output

The output for every scenario begins with a line containing "Scenario #i:", where i is the number of the scenario starting at 1. Then print a single line containing the shortest (and smallest) string as described above. Terminate the output for the scenario with a blank line.

Sample Input

1

2

TGCACA

CAT

Sample Output

Scenario #1:

TGCACAT

1. 问题描述与分析

给定一系列由字母表{A(腺嘌呤),C(胞核嘧啶),G(鸟嘌呤),T(胸腺嘧啶)}中字符构成的DNA串,找出包含所有这些串作为子串的最短串。如果有若干个这样的串,按字典法则取最小者。

为解此问题,先观察如下事实。

设$x[1..m]$、$y[1..n]$为两个字串。

(1) 若y是x的子串,则包含x、y的最短字串为x。若x是y的子串,则y是包含x、y的最短字串。

(2) 设y不是x的子串且x也不是y的子串,且令

$$l_1 = \max\{k \mid x[1..k] = y[n-k+1..n]\}, \quad l_2 = \max\{k \mid x[m-k+1..m] = y[1..k]\} \tag{11-2}$$

即 l_1 表示 x、y 的前、后缀的最长公共串的长度(见图 11-8(a)),l_2 表示 y、x 的前、后缀的最长公共长度(见图 11-8(b))。若 $l_1 > l_2$,则包含 x、y 的最短字串为 $y[1..n]+x[l_1+1..m]$,否则若 $l_2 > l_1$,为 $x[1..m]+y[l_2+1..n]$。如果 $l_1 = l_2$,则 $y[1..n]+x[l_1+1..m]$和 $x[1..m]+y[l_2+1..n]$都是包含 x、y 的最短字串。

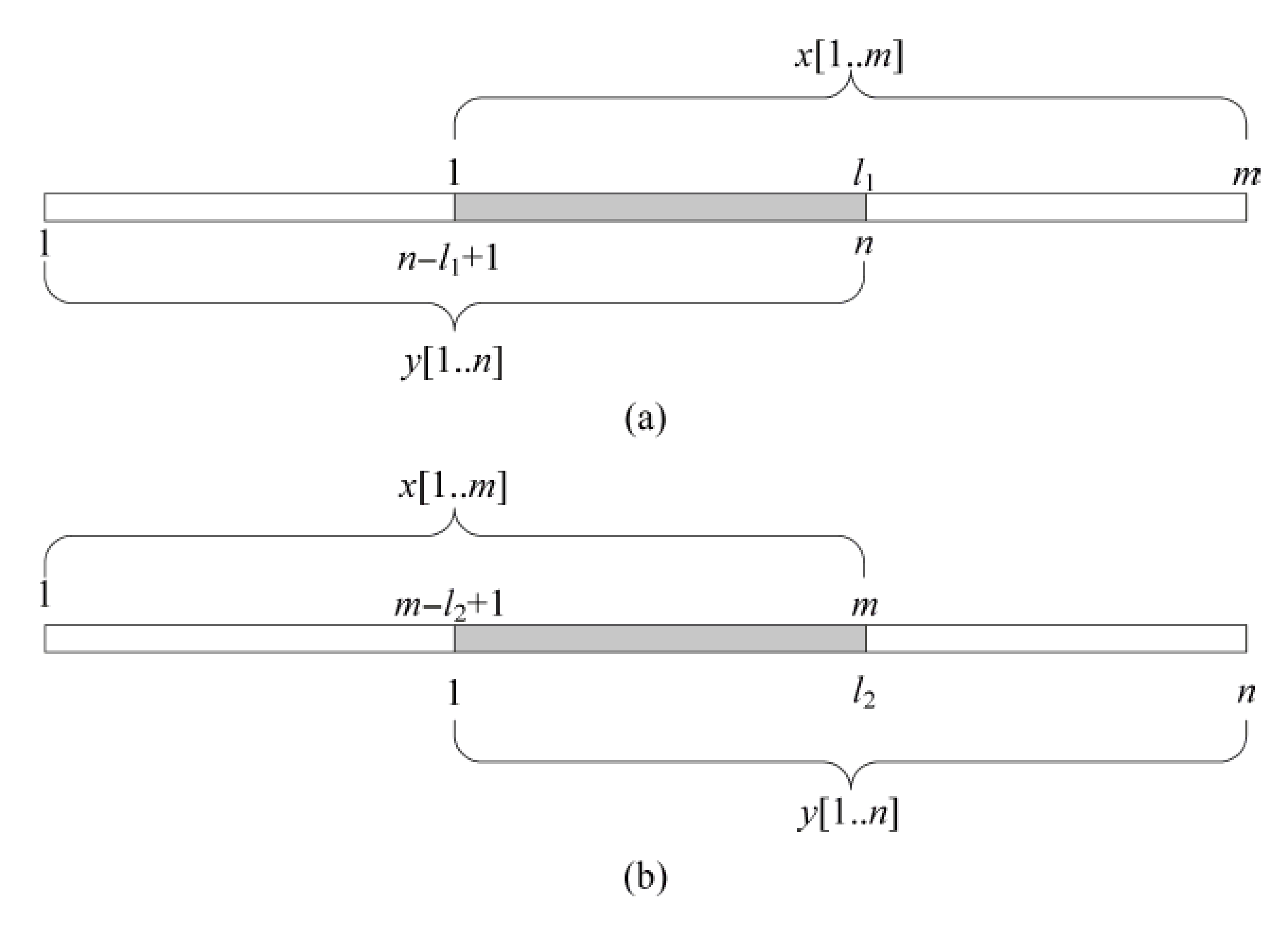


图 11-8　最长公共串的长度

下面以输入样例的数据为例说明此结论。此处 $x=$“TGCACA”,$y=$“CAT”,$m=6$,$n=3$。由于 $x[1..1]=y[3..3]=$“T”,$l_1=1$。$y[1..2]=x[5..6]=$“CA”,故 $l_2=2$。由于 $l_2=2>1=l_1$,所以以“TGCACA”和“CAT”为子串的最短字串是 $x[1..6]+y[3..3]=$“TGCACAT”。

2. 算法描述

根据上述结果,将计算包含两个字串 x、y 的最短字串过程描述如下。

```
SHORTESTSTRING(x, y)
if y 是 x 的子串
    then return x
if x 是 y 的子串
    then return y
l1 ← max{k | x[1..k] = y[n−k+1..n]}
l2 ← max{k | x[m−k+1..m] = y[1..k]}
if l1 < l2
    then return x[1..m] + y[l2+1..n]
    else if l2 < l1
        then return y[1..n] + x[l1+1..m]
return y[1..n] + x[l1+1..m]及 x[1..m] + y[l2+1..n]
```

算法 11-4　计算包含字符串 x、y 的最短字串过程

对本题中的一个案例,假定 n 个 DNA 串存放在数组 DNAS[1..n]中。我们用如下的贪婪策略解决此问题:current-solutions 从 DNAS[1]开始,依次计算包含 current-solutions

中串和 DNAS[i](i=2, 3, …, n)两者的最短子串,计算所得的 result 添加到 new-solutions 中,并用 new-solutions 更新 current-solutions。最终 current-solutions 的最小者即为所求,写成伪代码过程如下:

```
DNA-LABORARY(DNAS)
1 n←length[DNAS]
2 current-solutions←{DNAS[1]}
3 for i←2 to n
4      do new-solutions←∅
5         length←∞
6         for each string in current-solutions
7             do result←SHORTEST-STRING(string, DNAS[i])
8             if result 中串长不超过 length
9                then if result 中串长小于 length
10                       then length←result 中串长
11                            new-solutions←∅
12                     new-solutions←new-solutions ∪ result
13        current-solutions←new-solutions
14 SORT(current-solutions)                    /* 按字典顺序排序
15 return current-solutions[1]
```

算法 11-5 解决 DNA Laborary 的算法过程

这是一个渐增型算法,第 3～10 行的 **for** 循环维持的循环不变量是:在第 3 行和第 4 行的 for 循环的每次重复之初,*current-solutions* 是包含 *DNAS*[1]…*DNAS*[$i-1$]的最短子串集合。

利用此循环不变量,很容易证明此算法是正确的。

3. 程序实现

要实现算法 11-4,需要有一个计算式(11-2)中定义的串 x、y 的最长前后缀公共串长度的函数。

```
size_t pre_post_fix(char * x, char * y){/* 计算 max{k|x[1..k]=y[n-k+1..n]} */
    size_t m=strlen(x), n=strlen(y), l=m<n? m:n;
    char  * subx=(char * )malloc(l * sizeof(char)),
          * suby=(char * )malloc(l * sizeof(char));
    l--;
    while (l>0) {
        strncpy(subx, x, l), strncpy(suby, y+n-l, l);
        subx[l]=suby[l]=0;
        if(strcmp(subx, suby)==0)
            break;
        l--;
    }
    free(subx);free(suby);
```

```
14      return l;
15 }
```

程序 11-1　计算字符串 x、y 的前、后缀最长公共串长度的 C 函数

函数 pre_post_fix 从 x、y 的长度较小者 l 开始递减地逐一检测 x[0..l－1]与 y[n－l..n－1]是否相等，第一次检测到该条件成立时的 l 值即为所求(第 6～12 行的 **while** 循环)，第 14 行将其返回。利用该函数，可将算法 11-4 实现如下。

```
1 LinkedList * shortest_common_str(char * x, char * y){
2     size_t m=strlen(x), n=strlen(y);
3     char * r=(char * )calloc(m+n+1, sizeof(char));
4     LinkedList * L=createList(m+n+1,strcmp);          /* 创建字符串类型链表 */
5     if(strstr(x, y)){                                 /* y 是 x 的子串 */
6         listPushBack(L, x);
7     }else if(strstr(y, x)){                           /* x 是 y 的子串 */
8         listPushBack(L, y);
9     }else{
10        size_t l1=pre_post_fix(x, y), l2=pre_post_fix(y, x);
11        if(l1>l2){                                    /* 返回 y[1..n]+x[l1+1..m] */
12            strcpy(r, y);
13            strcpy(r+n, x+l1);
14            listPushBack(L, r);
15        }else if(l1<l2){                              /* 返回 x[1..m]+y[l2+1..n] */
16            strcpy(r, x);
17            strcpy(r+m, y+l2);
18            listPushBack(L, r);
19        }else               {/* 返回 y[1..n]+x[l1+1..m]和 x[1..m]+y[l2+1..n] */
20            strcpy(r, y);
21            strcpy(r+n, x+l1);
22            listPushBack(L, r);
23            strcpy(r, x);
24            strcpy(r+m, y+l2);
25            listPushBack(L, r);
26        }
27    }
28    return L;
29 }
```

程序 11-2　实现算法 11-4 的 C 函数

对程序 11-2 的说明如下。

(1) 由于返回值可能只有一个串(y[0..n－1]＋x[l1..m－1]或 x[0..m－1]＋y[l2..n－1])，也可能是两个串(y[0..n－1]＋x[l1..m－1]或 x[0..m]＋y[l2..n－1])，故用一个链表来存储返回值。因此，函数的返回值类型为链表指针 LinkedList *(2.1.3 节定义)。

(2) 第 5～9 行实现算法 11-4 中的第 1～4 行的操作。此处调用 C 的库函数 strstr 来检

测 x 是否为 y 的子串(第 5 行 strstr(x，y))或 y 是否为 x 的子串(第 7 行 strstr(y，x))。

(3) 第 9～27 行实现算法 11-4 中的第 5～10 行的操作。此处调用程序 11-1 中定义的函数 pre_post_fix，计算 x、y 的前后缀最长公共字串长度 l1 和 y、x 的前后缀最长公共字串长度 l2(第 10 行)。由于对不同情形下的返回值时存储在链表 L 中，所以统一地在第 28 行将 L 返回。

利用程序 11-2 定义的函数 shortest_common_str，将算法 11-5 实现如下。

```
char * dna_laborary(char * dnas[], int n){
    char * r=(char * )calloc(100, sizeof(char));
    LinkedList * currentSolution=createList(100, strcmp);
    listPushBack(currentSolution, dnas[0]);
    for(int i=1; i<n; i++){
        LinkedList * newSolution=createList(100, strcmp);      /* 设置 newSolution 为空集 */
        int length=200;                                         /* newSolution 中串的长度 */
        ListNode * a=currentSolution->nil->next;        /* a 指向 currentSolution 的首结点 */
        while (a!=currentSolution->nil){                /* 对 currentSolution 扫描 */
            char * string=a->key;                       /* 存储在 currentSolution 中当前串 */
            LinkedList * result=shortest_common_str(string, dnas[i]);
            ListNode * b=result->nil->next;
            if(strlen(b->key)<=length){
                if(strlen(b->key)<length){              /* r 中串比 newSolution 中串更短 */
                    length=strlen(b->key);
                    clrList(newSolution, NULL);
                    newSolution=createList(100, strcmp);
                }
                ListNode * c=result->nil->next;
                while(c!=result->nil) {                 /* 将 r 中串加入 newSolution */
                    listPushBack(newSolution, c->key);
                    c=c->next;
                }
            }
            clrList(result, NULL);
            a=a->next;
        }
        clrList(currentSolution, NULL);
        currentSolution=newSolution;              /* 用 newSolution 更新 currentSolution */
    }
    if(currentSolution->n>1)                      /* 不止一个最短串,排序 */
        listQuickSort(currentSolution, currentSolution->nil->next, currentSolution->nil);
    strcpy(r, currentSolution->nil->next->key);
    clrList(currentSolution, NULL);
    return r;
}
```

程序 11-3 实现算法 11-5 的 C 函数

程序 11-3 的说明如下。

（1）假定将问题输入中一个案例的数据（n 个 DNA 串）存储于一个字符串数组 dnas 中，其长度为 n。这两者作为函数 dna_laborary 的参数。该函数将返回包含所有 n 个 DNA 串的最短字串，故其返回值为 **char** * 。

（2）由于设计时不能确定集合 current-solution 和 new-solution 中串的个数，所以用链表 LinkedList 来表示这两个集合。第 5～30 行的 for 循环对应算法 11-5 中的第 3～13 行的 for 循环。其中，第 9～27 行的 **while** 循环对应算法 11-5 中第 6～12 行对 current-solution 的扫描。第 11 行调用程序 11-1 定义的函数 shortest_common_str，完成算法中第 7 行的操作。第 13～24 行对应算法中第 8～12 行的分支结构。其中第 19～23 行的 **while** 循环完成算法中第 12 行

$$new\text{-}solutions \leftarrow new\text{-}solutions \cup result$$

的操作。注意，对每一个链表，在舍弃之前均需调用 clrList 释放存储空间。

（3）第 31～32 行完成算法中第 13 行的操作，对得到的 currentSolution 调用函数 listQuickSort排序。程序中所有对链表的操作函数均定义于 2.1.3 节的程序中。

为节省篇幅，完成解决本问题的主函数在此不予罗列。连同程序 11-1 和 11-2 都存储于文件夹 chap11/DNA Laborary 中的源文件 DNALaborary. c，读者可打开对照研读。

11.2　最长回文子串问题

一个字符串 $P[1..m]$称为回文，指的是 $P[1..m]$关于其中心（若 m 为奇数，其中心为 $P[m/2]$，若 m 为偶数，则其中心为 $P[m/2]$与 $P[m/2+1]$之间的位置）对称。例如，ABCBA 是一个回文，而 ABCAB 就不是回文。我们的目标是在一个字符串 $S[1..n]$中查寻最长的回文子串 $P[1..m]=S[s+1..s+m]$。这个问题可以视为非固定模式匹配问题。因为相对于固定模式匹配，虽然也是在文本中查找一个特定的子串，但本问题中的模式事先并不知道其构成字符和长度。

如前所述，解决文本 $S[1..n]$中最长回文子串问题，需要对所有 $2n+1$ 个可能的中心，考察能得到回文子串，从中选出长度最大者。这 $2n+1$ 个中心分别是 $S[1]$、$S[2]$、…、$S[n]$（共有 n 个）和 $S[1]$之前的位置，$S[1]$、$S[2]$之间的位置、$S[2]$、$S[3]$之间的位置，…，$S[n-1]$、$S[n]$之间的位置和 $S[n]$后的位置（共有 $n+1$ 个）。为使对这两类中心所做的操作描述的一致性，可以针对 $S[1..n]$构造一个辅助串 $T[1..2n+1]$，其中 $T[1]$、$T[3]$、…、$T[2n+1]$填写一个不属于字符集 Σ 的字符，譬如记为＃、$T[2]$、$T[4]$、…、$T[2n]$分别填写 $S[1]$、$S[2]$、…、$S[n]$。这一操作可描述为以下算法过程。

```
GET-AUXSTR(S)
n←length[S]
allocate T[1..2n+1] and initialized to {＃, ＃, …, ＃}
for i←1 to n
    do T[2i]←S[i]
return T
```

算法 11-6　计算文本串 S 的辅助串的算法过程

例如,若文本 $S[1..4]$=ABCB,则对应的辅助字串 $T[1..9]$=#A#B#C#B#。为了计算 S 的最长回文子串,对 $1\leqslant i\leqslant 2n+1$ 计算 $\pi[i]$:T 中以 $T[i]$为中心最长回文子串的长度之一半。例如,对 $T[1..9]$=#A#B#C#D#,有 $\pi[1..9]=\{0,1,0,1,0,3,0,1,0\}$。在计算出来的数组 $\pi[1..2n+1]$中,假定最大者为 $\pi[t]=k$,若 $t=2i+1$,则以 $S[i]$、$S[i+1]$之间的位置为中心,长度为 k 的子串必为 S 中最长的回文。若 $t=2i$,则 S 中以$S[i]$为中心,长度为 k 的子串为 S 的最长回文子串。例如,对于上述 $S[1..4]$=ABCB,对应的 $T[1..9]$=#A#B#C#B#计算出来的 $\pi[1..9]$中的最大者为 $\pi[6]=3$。由于 $6=2\times 3$,所以 S 中以 $S[3]$为中心的长度为 3 的子串 $S[2..4]$=BCB 为 S 的最长回文子串。这样计算文本串 S 的最长回文子串就转换成计算对应的数组 $\pi[1..2n+1]$了。

11.2.1 强力算法

为计算文本串 $S[1..n]$对应的数组 $\pi[1..2n+1]$,先调用过程 GET-AUXSTR 构造出辅助串 $T[1..2n+1]$。显然,$T[1]=T[2n+1]=0$。对于每一个 $1<i<2n+1$,从 $k=1$ 开始,以 $T[i]$为中心通过检测 $T[i-k]$与 $T[i+k]$是否相等决定回文扩张是否继续,即 k 是否自增 1,直至 $T[i-k]\neq T[i+k]$。结束扩张时,记 $\pi[i]$为 $k-1$。写成如下的伪代码过程。

```
LANGEST-PALINDROM(S)
1 T(GET-AUXSTR(S)
2 n←length[T]
3 allocate π[1..n] and initialized to {0, 0, …, 0}
4 for i←2 to n-1
5     do k←1
6         while i-k>0 and i+k<n and T[i-k]=T[i+k]
7           do k←k+1
8         π[i]←k-1
9 return π
```

算法 11-7　计算文本串 S 的最长回文子串的强力算法

对算法 11-7 而言,最坏情形发生在 S 的所有字符均相等的时候。此时,对每一个 $2\leqslant i\leqslant n/2$,第 5~7 行的 **while** 循环将重复 i 次。因此,第 7 行要执行 $1+2+\cdots+(n/2-1)+n/2=(n-2)n/4+n/2$。式中的$(n-2)n/4$ 是运行时间是 $1+2+\cdots+(n/2-1)$的和。而对 $n/2<i<n$,根据对称性,第 7 行的操作也要执行 $1+2+\cdots+(n/2-1)=(n-2)n/4$ 次(见图 11-9)。因此,第 7 行要执行$(n-2)n/2+n/2=n(n-1)/2$ 次。于是 LANGEST-PALINDROM 的运行时间为 $\Theta(n^2)$。

图 11-9 展示了算法 LANGEST-PALINDROM 运行于串 S=AAAA 的过程。图 11-8(a)~图 11-8(g)表示算法第 4~8 行 **for** 循环的每次重复。每一部分中 i 指示处于当前中心位置,双向弧数目表示内嵌在第 6~7 行的 **while** 循环重复次数,也就是所要计算的 $\pi[i]$。图 11-8(h)表示计算所得的数组 $\pi[1..9]$。由于串 S 内的所有字符都是一样的,故对于每一个 i,以 $T[i]$为中心向左右作扩张都能达到"极限"状态 $i-k<1$ 或 $i+k>n$ 才停止。故此例表示出算法所遭遇的最坏情形。

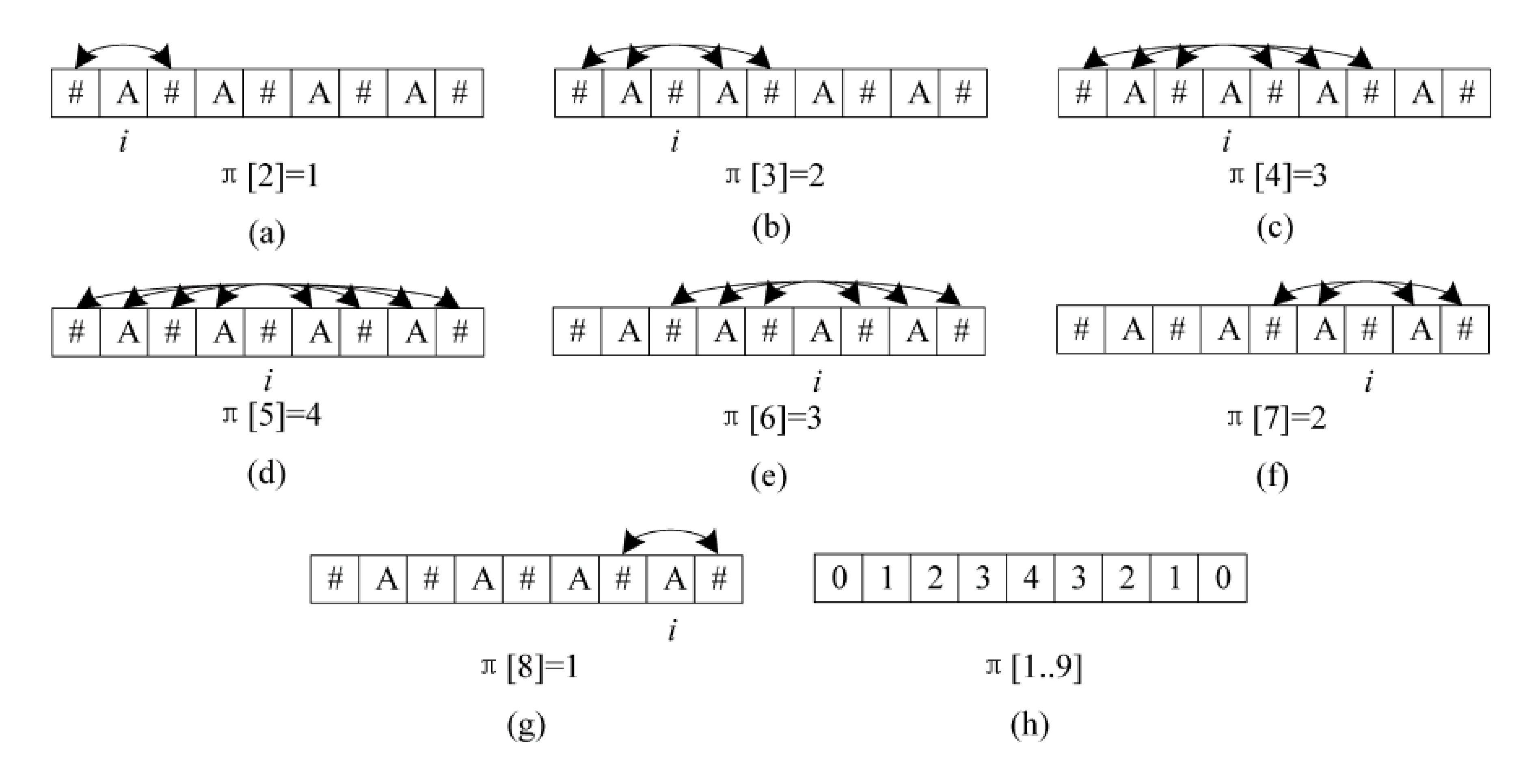


图 11-9　对 S=AAAA 运行 LANGEST-PALINDROM(S)的过程

此例还展示出回文串的一个诱人特性，注意以 $T[5]$为中心的回文子串最长，它涵盖了整个 T。也就是说 T 本身是一个回文串，根据回文串的对称性，包含在 T 内部的回文子串关于 $T[5]$是对称的。这一事实反映在数组 π 上，元素的值关于 $\pi[5]$是对称的。换句话说如果事先知道 $T[5]$为中心的回文子串涵盖了整个 T，则对 $6\leqslant i<9$，记 i'为 i 关于 5 的对称位置，则必有 $\pi[i]=\pi[i']$，无须再以 $T[i]$为中心进行“扩张”了。在 11.2.2 节中，会看到这一特性是改善计算最长回文子串的关键。

11.2.2　Manacher 算法

根据上文的对一个特例的讨论，已知可以利用回文关于中心对称的特性改善算法的运行时间效率。现在对一般的情形进行形式化的表述。对于任一文本串 $S[1..n]$，其对应的辅助串设为 $T[1..2n+1]$。如前所述，有 $\pi[1]=\pi[2n+1]=0$。由于 $\pi[2]$对应的是以 $T[2]$为中心的最长回文子串区间半径，而 $T[2]$左边仅有一个特殊字符 #，而右边也有一个 #，故 $\pi[2]=1$。对于 $2<i<2n+1$，假定 $\pi[1..i-1]$已经算得，我们来计算 $\pi[i]$。我们知道，对 $0<k<i$，$T[k-\pi[k]..\ k+\pi[k]]$是 T 中以 k 为中心的最长回文子串。$k-\pi[k]$，$k+\pi[k]$分别是该回文子串的左、右边界。选取右边界最大的那个回文子串，记其中心为 C，右边界 $R=C+\pi[C]$，左边界 $L=C-\pi[C]$，则 $T[L..R]$是 T 中以 C 为中心的最长回文子串(见图 11-10)。

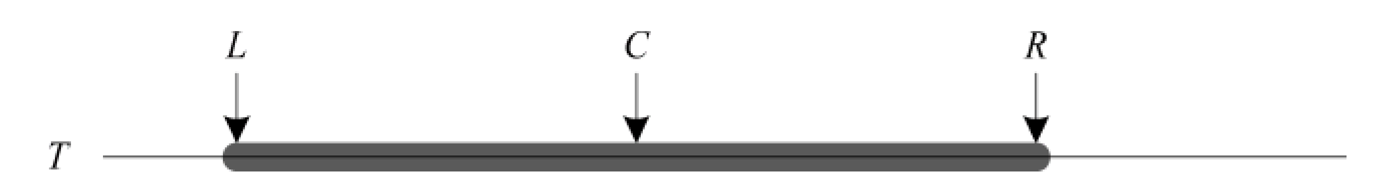


图 11-10　T 中以 C 为中心，左右边界分别为 L、R 的回文子串

利用 $T[L..R]$关于 C 的对称性，我们有如下观察。

(1) 若 $i<R$，即 $T[i]$位于 $T[L..R]$内。令 i'为 i 关于 C 的对称点，必有 $L<i'\leqslant C(<i)$，当然 $\pi[i']$是已知的。根据 $\pi[i']$的定义，有 $T[i'-\pi[i']..\ i'+\pi[i']]$是以 i'为中心的最长回

文子串。

① 若 $i'-\pi[i']\geq L$(因此也有 $i+\pi[i']\leq R$),则 $T[i'-\pi[i']..\ i'+\pi[i']]$含于 $T[L..R]$内。根据 $T[L..R]$的对称性,$T[i-\pi[i']..\ i+\pi[i']]$也包含在 $T[L..R]$内,且是以 i 为中心的回文子串(未必最大)。所以,$\pi[i]$从 $\pi[i']$开始取值(见图 11-11)。

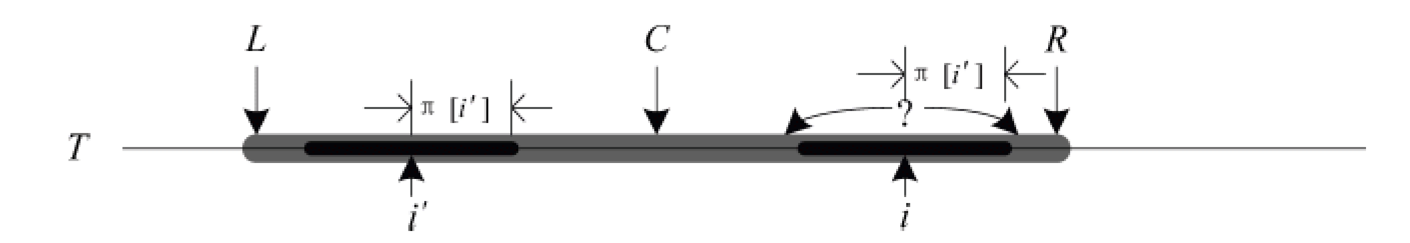


图 11-11 当前扩张中心 $T[i]$及其关于 $T[C]$的对称点 $T[i']$均位于 $T[L..R]$内且以 $T[i']$为中心的最长回文子串含于 $T[L..R]$内

② 若 $i'-\pi[i']<L$(因此也有 $i-\pi[i']>R$)我们只能保证 $T[L..\ 2i'-L]$含于$T[L..R]$内且为回文子串(未必最大)而其在 $T[L..R]$中关于 C 对称的 $T[2i-R..R]$是以 i 为中心的回文。因此,$\pi[i]$应该从 $R-i$ 起取值(见图 11-12)。

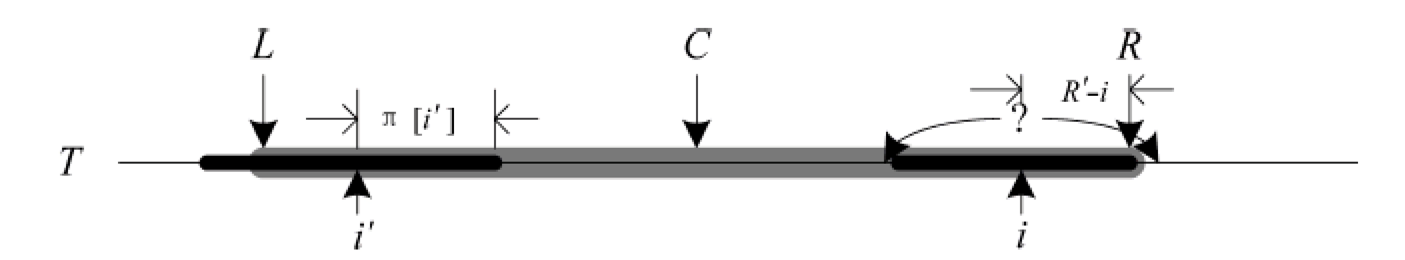


图 11-12 当前扩张中心 $T[i]$及其关于 $T[C]$的对称点 $T[i']$均位于 $T[L..R]$内,但以 $T[i']$为中心的最长回文子串的左边界位于 $T[L]$的左边

也就是说,在 $i<R$ 的条件下,“扩张”的起点应该是 $\pi[i]=min\{\pi[i'],\ R-i\}$,$k=\pi[i]+1$,依次检测 $T[i-k]$是否等于 $T[i+k]$而决定 $\pi[i]$的增量。

(2) 若 $i\geq R$,即 $T[i]$位于 $T[L..R]$中心的右边,此时,$T[L..R]$的对称性对以 i 为中心的回文长度的确定不能提供任何有用的信息(自然,$\pi[0..i-1]$也都是如此)。所以,$\pi[i]$只能从 0 起,k 从 1 起,依次检测 $T[i-k]$是否等于 $T[i+k]$($k=1, 2,\cdots$)而决定 $\pi[i]$的增量。

对当前中心位置的“扩张”结束时,若 $i+\pi[i]>R$,则将 C 更新为 i,而 R 自然就要更新为 $i+\pi[i]$(L 自然为 $i-\pi[i]$),为计算下一个 i 做好准备。

```
MANACHER(S)
T←GET-AUXSTR(S)
n←length[T]
π[1]←π[n]←0,π[2]←1
C←2,R←i←3,k←π[i]+1
while i<n
    do if T[i-k]=T[i+k] ▷还在“扩张”中
          then π[i]←π[i]+1
               k←k+1
          else if i+π[i]>R ▷“扩张”结束且需要调整新的 C、R
                  then C←i
                       R←i+π[i]
               i←i+1,i'←2C-i ▷开始新的中心位置并计算其关于 C 的对称位置
               if i<R ▷情形 1
```

```
                  then π[i]←min{R−i, π[i′]}
                  else π[i]←0 ▷情形 2
              k←π[i]+1
return π
```

算法 11-8　线性时间的计算最长回文子串算法

图 11-12 展示了算法 11-8 运行于文本串 S=BABCBABCBACC 的过程。

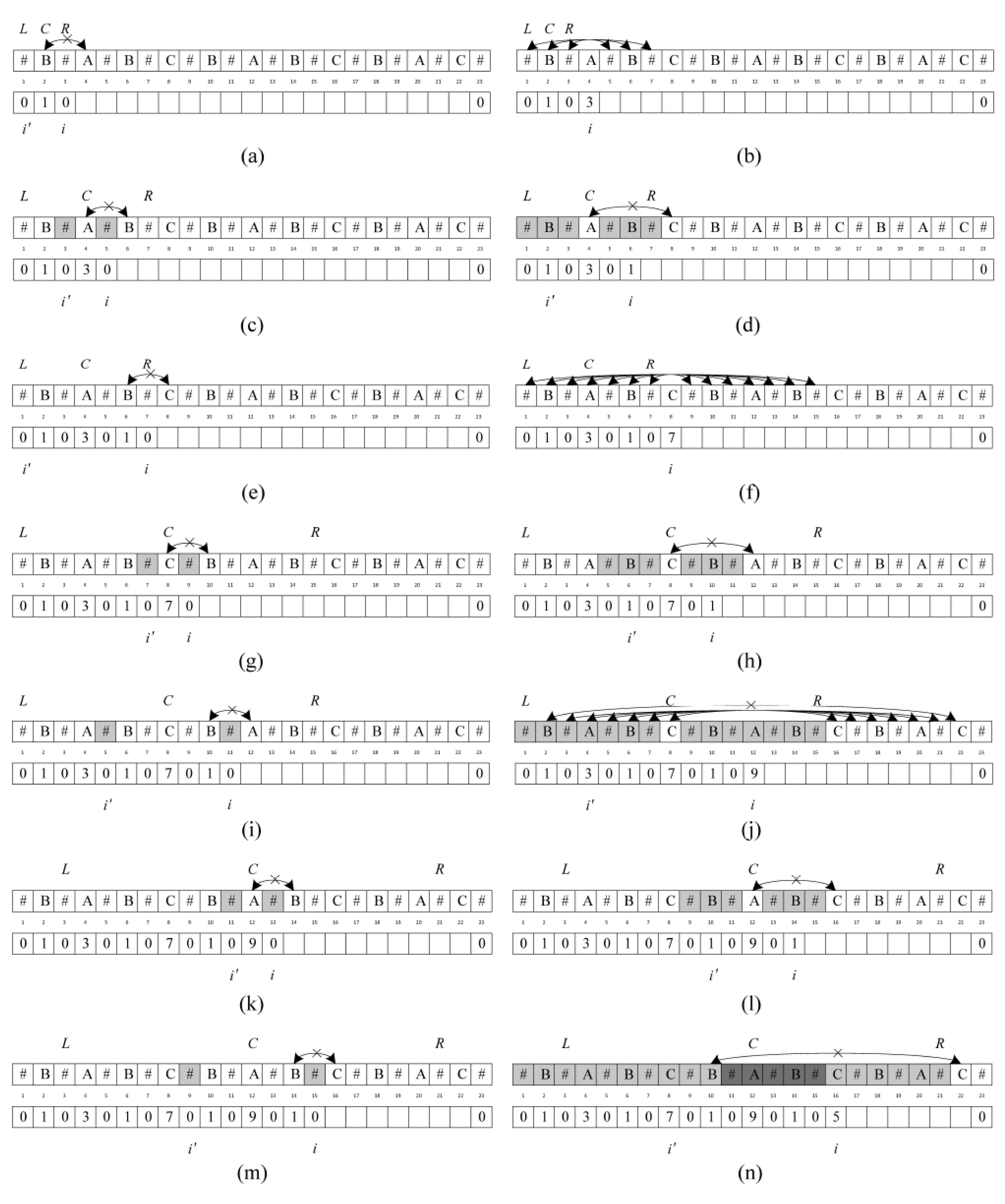


图 11-13　算法 11-6 运行于文本串 S=BABCBABCBACC 的过程

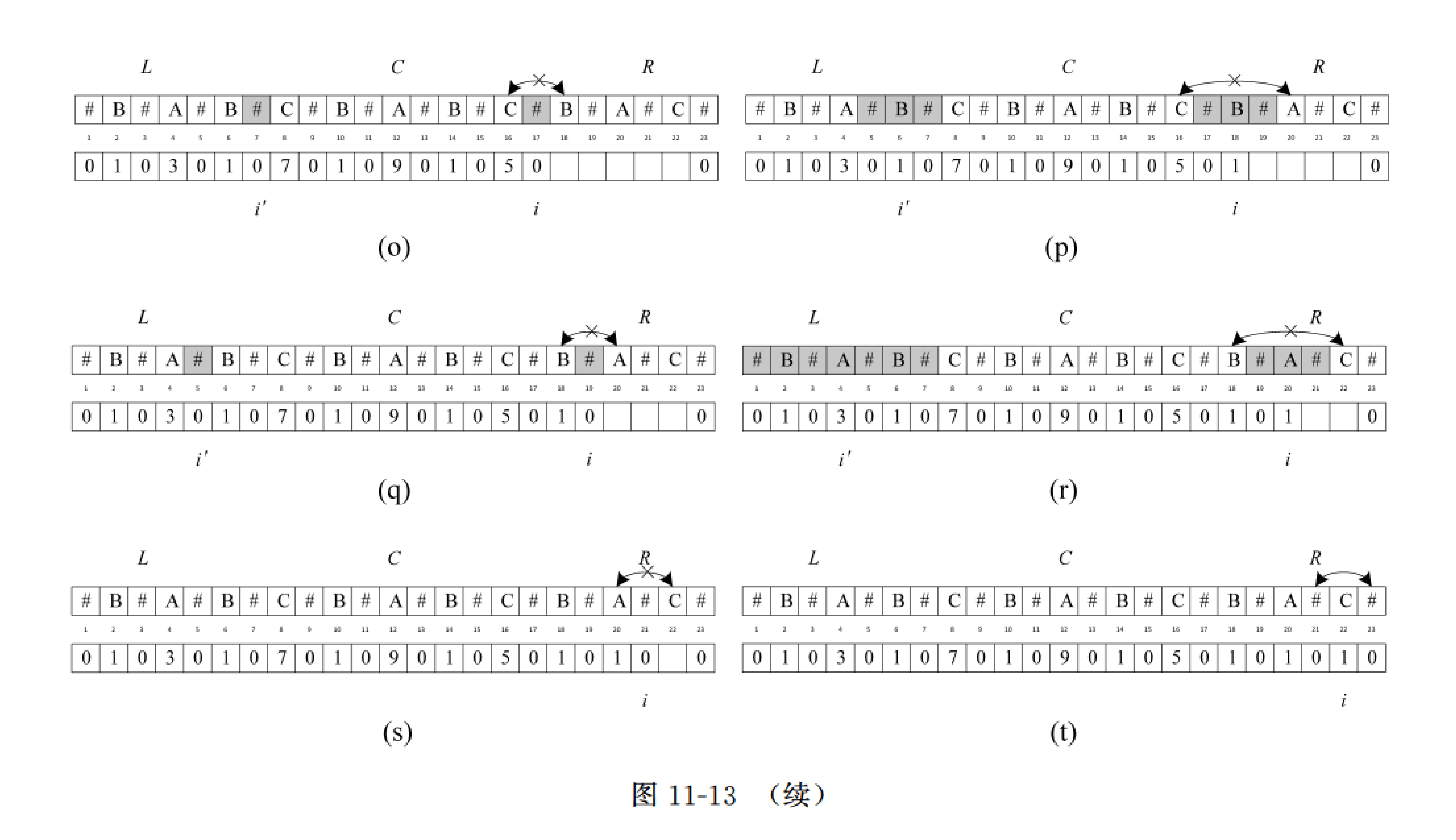


图 11-13 (续)

图 11-13(a)初始时,$C=2$,由于 $\pi[C]=\pi[2]=1$,故 $R=C+\pi[C]=3$,相应地,R 关于 C 的对称位置 L 为 1。要计算的是 $i=3$ 处的 $\pi[i]$,由于 $i=3=R$,属于上述的情形 2,所以 $\pi[i]$初始化为 0,k 初始化为 $\pi[i]+1=1$。由于 $T[i-k]=T[2]=B\neq A=T[4]=T[i+k]$(图中用带×的双向弧线表示),于是 $\pi[i]$保持为 0,本轮"扩张"结束。由于 $i+\pi[i]=3=R$,故 C、L、R 保持不变。

图 11-13(b)中,上一轮"扩张"结束后,i 自增 1,i 为 4,大于 R。仍属情形 2,于是 $\pi[i]$从 0 开始,k 从 $\pi[i]+1=1$ 起进行扩张,扩张直至超出文本左边界为止。每次扩张 $\pi[i]$自增 1,累计为 3。由于 $i+\pi[i]=4+3=7>R$,调整 C 为 $i=4$,$R=i+\pi[i]=7$,L 与 R 关于 C 对称,自然就应该为 1。

图 11-13(c)中,本轮的扩展中心 $i=5<7=R$,属情形 1。i 关于 C 的对称点 $i'=3$,$\pi[i']=0$,$i'-\pi[i']=3\geqslant 1=L$,故 $\pi[i]$初始化为 $\pi[i']$,扩张从 $k=\pi[i]+1=1$ 开始,但是一开始便遭遇了失败,扩张戛然停止。于是 $\pi[i]$保持为 0。$i+\pi[i]=5<7=R$,故 C、L、R 保持不变。

图 11-13(d)中,i 来到 $6<R$,仍属情形 1。i 关于 C 的对称点 $i'=2$,与图 11-13(c)中一样,$i'-\pi[i']=1\geqslant 1=L$,故 $\pi[i]$初始化为 $\pi[i']$,扩张从 $k=\pi[i]+1=2$ 开始。扩张的结果是 $\pi[i]=1$。和图 11-13(c)中一样,C、L、R 保持不变。

图 11-13(e)中,i 自增 1 后为 $7=R$。和图 11-13(a)中一样,属情形 2。同样的以 i 为中心的扩张结果为 $\pi[i]=0$。由于 $i+\pi[i]=R$,故 C、L、R 不变。

图 11-13(f)中,本轮扩张中心位于 $i=8>R$,属情形 2,从 $\pi[i]=0$ 开始扩张。与图 11-13(b)类似,本轮扩张的结果 $\pi[i]=7$。由于 $i+\pi[i]=15>R$,故 C、L、R 分别调整为 8、1、15。

图 11-13(g)~图 11-13(j) 分别表示在 C、L 和 R 分别为 8、1、15 时,扩张中心 i=9、10、11、12 的"扩张"操作。由于 i 取这些值时均有 $i<R$,故都属于情形 1。和图 11-13(c)一样,

这些 i 值对应的关于 C 的对称点 i' 及其 $\pi[i']$ 还都满足 $i'-\pi[i']\geqslant L$，故每次扩张，都是从 $\pi[i]=\pi[i']$，$k=\pi[i]+1$ 开始检测 $T[i-k]$ 与 $T[i+k]$ 是否相等。图 11-13(j)操作完毕时，$i+\pi[i]=21>15=R$。于是，C、L 和 R 分别调整为 12、3 和 21。

与图 11-13(g)～图 11-13(j)相仿，图 11-13(k)～图 11-13(r)在 C、L 和 R 保持不变(分别为 12、3、21)的情形下，扩张中心为 $i=13$、14、…、20(均小于 R)时的扩张操作。虽然均属于情形 1，但在图 11-13(k)、图 11-13(l)、图 11-13(m)、图 11-13(o)、图 11-13(p)、图 11-13(i)和图 11-13(q)中，i 关于 C 的对称点，对应的 $i'-\pi[i']$ 均不小于 L，所以扩张起点 $\pi[i]$ 为 $\pi[i']$。而(n)和(r)中，$i'-\pi[i']$ 小于 L，所以扩张起点 $\pi[i]$ 为 $R-i$。图 11-13(r)扩张完成时 $\pi[i]=1$，$i+\pi[i]=21=R$，故 C、L 和 R 保持不变。

图 11-13(s)和图 11-13(t)中，$i=21$ 和 $i=22$ 时，$i\geqslant R$，属情形 2，扩张起点 $\pi[i]=0$，扩张结果 $\pi[i]$ 分别 0 和 1。

算法 11-6 的运行时间取决于第 5～15 行的 **while** 循环的重复次数。该循环的重复条件是 $i<n$，循环体是一个二分支结构。分支之一的第 9～15 中执行 $i\leftarrow i+1$ 的自增操作，所以这一分支必执行 $\Theta(n)$。第 7～8 行构成的另一个分支是以当前的 i 为中心，进行回文扩张。由于当前扩张过程中比较 $T[i-k]$ 和 $T[i+k]$ 相等的次数记录在 $\pi[i]$，成为后来与 i 关于某个 C 对称点的扩张起点，所以比较 $T[i-k]$ 和 $T[i+k]$ 相等的次数也是 $\Theta(n)$。

11.2.3　程序实现

C 语言没有向程序员提供计算字符串中最大回文子串的库函数。这里来实现线性运行时间的 MANACHER 算法。首先，需要实现算法 11-6 中计算辅助字串的 GET-AUXSTR 过程。

```
1 static char * getAuxStr(char  * s) {
2     int n=(int)strlen(s);
3     if (n==0) return NULL;
4     char  * t=(char * )calloc(2 * n+2, sizeof(char));
5     memset(t,'#', 2 * n+1);
6     for (int i=0; i <n; i++)
7         t[2 * i+1]=s[i];
8     return t;
9}
```

程序 11-4　实现算法 11-6 的 C 函数

函数 getAuxStr 计算由参数 s 传递给它的字符串的辅助字串 t，并作为函数值返回。程序代码与算法的伪代码结构上几乎是一致的。需要注意的细节如下。

(1) 若 s 是空串，辅助串当然亦为空。

(2) 第 4 行为辅助串 t 分配空间为 $2n+2$ 个字符，而非 $2n+1$ 个字符。这是因为 C 语言的字符串式存储在字符数组中以\0 为结束标志的字符序列。因此，需要为\0 预留存储空间。由于我们使用 calloc 函数为 t 分配空间，该函数在分配空间后将所有字节初始化为 0，

所以第 5 行调用库函数 memset 用#填充 $t[0..2n]$,$t[2*n+1]$自然就保留为\0。

(3) getAuxStr 仅仅在以下实现算法 11-8 的函数中被调用,并不需要(也不希望)其他代码无意中调用它而造成混乱,所以将其定义为 **static** 函数,这样就将其可见范围限制在本文件中了。

利用程序 11-4,可以将算法 11-8 实现如下。

```
1 int *manacher(char *s){
2     char *t=getAuxStr(s);
3     size_t n=strlen(t);
4     int *pi=(int *)calloc(n, sizeof(int));
5     int c=1, r=2, i=2, k=1;
6     pi[1]=1;
7     while (i<n-1) {
8         if (i+k<n&&t[i-k]==t[i+k]){           //以 i 为中心,扩展
9             pi[i]++;
10            k++;
11            continue;
12        }
13        if (i+pi[i]>r) {                      //调整参照回文子串
14            c=i;
15            r=i+pi[i];
16        }
17        i++;                                  //下一个扩展中心 i
18        int i1=2*c-i;                         //i 关于参照回文子串中心 c 对称位置
19        if (r>i)                              //i 在参照回文子串内部
20            pi[i]=min(r-i, pi[i1]);           //pi[i]的初始值
21        k=pi[i]+1;                            //扩展的起点
22    }
23    free(t);
24    return pi;
25 }
```

程序 11-5 实现算法 11-8 的 C 函数

程序 11-5 的代码结构与算法 11-8 的伪代码结构也是高度一致的。需要注意的是第 20 行调用的计算 $r-i$ 和 pi[i1]的较小值的函数,代码如下。

```
int min(int a, int b){
    return a<b? a:b;
}
```

此外,由于用 calloc 函数为 pi 分配的空间,所以 pi[0..n-1]均初始化为 0。这样,对发生情形 2(即 i≥r)时无须将 pi[i]重置为 0。程序 11-4 和程序 11-5 的代码均存储在文件夹 utility 中的源文件 textmatch. c 中。

11.2.4　应用

Palindrome

The "U. S. Robots" HQ has just received a rather alarming anonymous letter. It states that the agent from the competing《Robots Unlimited》has infiltrated into "U. S. Robotics".《U. S. Robots》security service would have already started an undercover operation to establish the agent's identity, but, fortunately, the letter describes communication channel the agent uses. He will publish articles containing stolen data to the "Solaris" almanac. Obviously, he will obfuscate the data, so "Robots Unlimited" will have to use a special descrambler ("Robots Unlimited" part number NPRx8086, specifications are kept secret).

Having read the letter, the "U. S. Robots" president recalled having hired the "Robots Unlimited" ex-employee John Pupkin. President knows he can trust John, because John is still angry at being mistreated by "Robots Unlimited". Unfortunately, he was fired just before his team has finished work on the NPRx8086 design.

So, the president has assigned the task of agent's message interception to John. At first, John felt rather embarrassed, because revealing the hidden message isn't any easier than finding a needle in a haystack. However, after he struggled the problem for a while, he remembered that the design of NPRx8086 was still incomplete. "Robots Unlimited" fired John when he was working on a specific module, the text direction detector. Nobody else could finish that module, so the descrambler will choose the text scanning direction at random. To ensure the correct descrambling of the message by NPRx8086, agent must encode the information in such a way that the resulting secret message reads the same both forwards and backwards.

In addition, it is reasonable to assume that the agent will be sending a very long message, so John has simply to find the longest message satisfying the mentioned property.

Your task is to help John Pupkin by writing a program to find the secret message in the text of a given article. As NPRx8086 ignores white spaces and punctuation marks, John will remove them from the text before feeding it into the program.

Input

The input consists of a single line, which contains a string of Latin alphabet letters (no other characters will appear in the string). String length will not exceed 1000 characters.

Output

The longest substring with mentioned property. If there are several such strings you should output the first of them.

Sample

Input	output
ThesampletextthatcouldbereadedthesameinbothordersArozaupalanalapuazorA	ArozaupalanalapuazorA

这是一个很直接的题目,就是计算输入的字符串 s[0..n−1]的第一个(可能有多个)最长回文子串,并输出。调用程序 11-5 的 manacher 函数,返回的是关于 s 的辅助字串 t[0..2n+1]中每个字符为中心的最长回文子串长度的一半构成的数组 pi[0..2n+1]。用下列函数找出第一个最大值元素下标。

```
int maxele(int *a, int n){
    int m=0;
    for(int i=1; i<n; i++)
        if(a[i]>a[m])
            m=i;
    return m;
}
```

设调用 maxele(pi)返回值为 i,pi[i]的值为 length。于是,s[i/2−length/2..i/2+length/2−1]就是所求。写成代码如下。

```
char *longestPalindromeString(char *s){
    int n=strlen(s), *pi=manacher(t), i, length;
    char *s1=(char*)calloc(n, sizeof(char));
    n=2*n+1;
    i=maxele(pi, n);
    length=pi[i];
    memcpy(s1, s+i/2-length/2, length);
                    /*将 s[i/2-length/2..i/2+length/2-1]复制到 s1[0..length-1]*/
    free(pi);
    return s1;
}
```

完整的程序代码存储于 chap11/Palindrome 的源文件 palindrome.c,读者可打开研读。

11.3 近似匹配

在信息处理应用中,往往需要进行文本的近似匹配。例如,要在一段文本 T 中查找一个单词 W,但并不能完全确定 W 的正确拼写。在这种情况下,我们要求在 T 中查找与 W 最相近的单词。

11.3.1 最小编辑距离

首先,需要定义 T 中与 W 相近的单词的意义。为此,我们要明确将一个词 $P\in\Sigma^*$ 换成

另一个词 $W\in\Sigma^*$ 允许做的所有操作。这些操作包括如下。

（1）用一个字符替代另一个字符。

（2）删除一个字符

（3）插入一个字符

设 $P, W\in\Sigma^*$。将 P 变换成 W 过程中，所做的上述合法操作的次数定义为 P 和 W 的编辑距离，记为 $\text{dist}(P, W)$。显然，对两个给定的 $P, W\in\Sigma^*$ 而言，用合法操作将 P 变换成 W 的方式并非唯一。例如，假定 P=ghost，W=house，可以按下列步骤变换。

① ghost　删除第一个字符 g→　host

② host　在第二、三个字符之间插入字符 u→　houst

③ houst　将最后字符 t 替换成→e　house

在此方案中，$\text{dist}(P, W)=3$。

也可以将 P 中的所有字符都删除（进行了 $length[P]$次操作），然后逐一添加 W 中的各字符（进行 $length[W]$次操作）完成替换。这样，$\text{dist}(P, W)=length[P]+length[W]=10$。

由此可见，要将给定的字符串 P，用合法操作变换成指定字符串 W，存在多个可能的解决方案，每个方案都有一个编辑距离。也就是说，按上述定义的编辑距离不是唯一的。我们的目标是找出编辑距离最小者作为唯一值。显然，这是个组合优化问题（见本书 7.4.1 节）。可以利用第 8 章引入的“动态规划”策略解决这个问题。为此，我们考察本问题如下的最优子结构。

设 $P=P[1..n]$，$W=W[1..m]\in\Sigma^*$。考虑 P 的前缀 $P[1..i]$，$i=0, 1, \cdots, n$ 和 W 的前缀 $W[1..j]$，$j=0, 1, \cdots, m$ 的最小编辑距离。当 $i=0$ 时，$P[1..i]=\varepsilon(\in\Sigma^*)$，$P[1..i]$转换成 $W[1..j]$只有一种方案，那就是在 P（从空串 ε 开始）中逐一插入 $W[1..j]$的每个字符，所以，编辑距离（也是最小编辑距离为 j）。同样，对于 $j=0$，$W[1..j]=\varepsilon$。$P[1..i]$转换成 $W[1..j]$也只有一种方案：在 P 中逐一删除每一个字符，其最小编辑距离为 i。对于 $i>0$ 和 $j>0$ 的情况，可以用下列 3 种方式之一将 $P[1..i]$转换成 $W[1..j]$。

（1）删掉 $P[i]$，将 $P[1..i-1]$转换成 $W[1..j]$。

（2）在 $P[1..i]$后插入 $W[j]$，将 $P[1..i]$转换成 $W[1..j-1]$。

（3）$P[i]=W[j]$，直接将 $P[1..i-1]$转换成 $W[1..j-1]$，否则先将 $P[i]$换成 $W[j]$，再将 $P[1..i-1]$转换成 $W[1..j-1]$。

若设 d_{ij} 表示 $P[1..i]$转换成 $W[1..j]$的最佳方案所执行的合法操作次数，则根据以上分析，有如下的递归方程：

$$d_{ij}=\begin{cases} i & j=0 \\ j & i=0 \\ \min\{d_{i-1j-1}, d_{i-1j}+1, d_{ij-1}+1\} & i>0, j>0, P[i]=W[j] \\ \min\{d_{i-1j-1}+1, d_{i-1j}+1, d_{ij-1}+1\} & i>0, j>0, P[i]\neq W[j] \\ m & \end{cases} \tag{11-3}$$

这样，就可以构建一个自底向上的记表计算 d_{nm} 的算法。

```
OPTIMAL-EDIT-DISTANCE(P, W)
1 n←length[P], m←length[W]
2 allocate d[0..n, 0..m]
```

```
3 for j←0 to m
4       do d[0, j]←j
5 for i←1 to n
6       do d[i, 0]←i
7 for i←1 to n
8       do for j←1 to m
9             do a←min{d[i−1, j]+1, d[i, j−1]+1}
10               if P[i]=W[j]
11                 then b←d[i−1, j−1]
12                 else b←d[i−1, j−1]+1
13               d[i, j]←min{a, b}
13 return d[n, m]
```

算法 11-9 计算字符串 P、W 最小编辑距离的算法

很容易看出来,算法 11-9 的运行时间主要耗费在第 7～11 行的两重循环上,这两个嵌套的循环都是计数循环,外层重复 n 次,里层重复 m 次。循环体(第 9～13 行)内的操作都是常数时间的,因此,该算法的运行时间为 $\Theta(mn)$。下文中以 dist(P,W)表示 P、W 的最小编辑距离。

11.3.2 最佳近似匹配

现在来描述文本的最佳近似匹配问题。

输入:文本串 $T[1..n]\in\Sigma^*$,模式串 $P[1..m]\in\Sigma^*$。其中,$m\leqslant n$。

输出:字符串 $W[1..l]$,整数 $1\leqslant i<j\leqslant n$,使得 $j-i+1=l$,$W[1..l]=T[i..j]$且 dist(P, W)最小。

解决此问题的强力算法,是对所有的偏移量 $0\leqslant s<n$,和所有可能的长度 $1\leqslant l\leqslant n-s$ 利用 OPTIMAL-EDIT-DISTANCE 过程,计算 dist($P[1..m]$,$T[s+1..s+l]$),跟踪最小者对应的偏移量 index 和长度 length。伪代码过程描述如下。

```
NAIVE-APPROXIMATE-MATCHER(T, P)
1 n←length[T], m←length[P]
2 min-distance←∞
3 for s←0 to n−1
4       do for l←1 to n−s
5             do d←OPTIMAL-EDIT-DISTANCE(P, T[s+1..s+l])
6                if d<min-distance
7                  then min-distance←d
8                       index←s
9                       length←l
10 return index and length
```

算法 11-10 在文本 T 中查找模式 P 的最佳近似匹配的强力算法

算法 11-10 的第 3～9 行的两重循环的循环体(第 5～9 行)共重复 $O(n^2)$,每次重复,第

5 行调用 OPTIMAL-EDIT-DISTANCE 过程计算 dist$(P[1..m], T[s+1..s+l])$，耗时 $O(nm)$。故总的运行时间为 $O(mn^3)$。

其实，这也是个组合优化问题。考虑模式的前缀 $P[1..i]$与文本前缀 $T[1..j]$的所有后缀 $T[l..j]$$(1\leqslant l\leqslant j)$的编辑距离的最小者。对于 $i=0$，$P[1..i]=\varepsilon(\in\Sigma^*)$，我们认为 $P[1..i]$与 $T[1..j]$中的后缀 ε 的编辑距离为 0，是最小的。对于 $j=0$，$T[1..j]=\varepsilon(\in\Sigma^*)$，必须删掉 $P[1..i]$所有字符才能与之匹配。而对于 $i>0$ 且 $j>0$，和计算两个字符串的编辑距离相仿，可以用以下 3 种方式之一，得到 $P[1..i]$与 $T[1..j]$的所有后缀 $T[l..j]$$(0\leqslant l\leqslant j)$编辑距离的最小者。

(1) 删掉 $P[i]$，计算 $P[1..i-1]$与 $T[1..j]$的所有后缀的编辑距离的最小者。

(2) 在 $P[1..i]$后插入 $T[j]$，计算 $P[1..i]$与 $T[1..j-1]$的所有后缀的编辑距离的最小者。

(3) 若 $P[i]=W[j]$，直接计算 $P[1..i-1]$与 $T[1..j-1]$的所有后缀的编辑距离最小者，否则先将 $P[i]$换成 $T[j]$，再计算 $P[1..i-1]$与 $T[1..j-1]$ 的所有后缀的编辑距离最小者。

记 $P[1..i-1]$与 $T[1..j]$的所有后缀的的最小编辑距离值为 ad_{ij}，上述分析可表示为如下递归方程：

$$ad_{ij}=\begin{cases}0 & i=0\\ i & j=0\\ \min\{ad_{i-1j-1}, ad_{i-1j}+1, ad_{ij-1}+1\} & i>0, j>0, P[i]=T[j]\\ \min\{ad_{i-1j-1}+1, ad_{i-1j}+1, ad_{ij-1}+1\} & i>0, j>0, P[i]\neq T[j]\end{cases}\qquad(11\text{-}4)$$

据此，可以用记表计算的方式计算出 $P[1..i]$与 $T[1..j]$的所有后缀 $T[l..j]$$(0\leqslant l\leqslant j)$的最小编辑距离 ad_{mj}，$j=1, 2, \cdots, n$。

```
APPROXIMATE-MATCHER(P, T)
1 m←length[P], n←length[T]
2 allocate ad[0..m, 0..n]
3 for j←0 to n
4       do d[0, j]←0
5 for i←1 to m
6       do d[i, 0]←i
7 for i←1 to m
8       do for j←1 to n
9              do a←min{ad[i-1, j]+1, ad[i, j-1]+1}
10                if P[i]=T[j]
11                   then b←ad[i-1, j-1]
12                   else b←ad[i-1, j-1]+1
13                ad[i, j]←min{a, b}
13 return ad
```

算法 11-11　计算文本 T 中模式 P 的近似匹配的动态规划算

算法 11-11 在 $\Theta(mn)$时间(第 7～13 行两重循环的重复次数)内按式(11-4)计算出数表 $ad[0..m, 0..n]$。在 ad_{m1}、ad_{m2}、…、ad_{mn} 中找出最小者 ad_{mt}，就得到了 P 在 T 中的最佳

近似匹配的最后一个字符的下标 t。可以用下列的过程得到最佳近似匹配 $T[l..t]$。

```
SOLUTION(T, ad, i, j)
1 if i=0
2     then return
3 if ad[i, j]=ad[i-1, j-1] or ad[i, j]=ad[i-1, j-1]+1
4     then SOLUTION(T, ad, i-1, j-1)
5          print T[j]
6    else if ad[i, j]=ad[i, j-1]+1
7             then SOLUTION(T, ad, i, j-1)
9                  print T[j]
10            else SOLUTION(T, ad, i-1, j)
```

算法 11-12 根据 APPROXIMATE-MATCHER(P, T)算得的数表 ad 计算最佳匹配的算法

算法 11-12 是一个递归过程,顶层调用应该是 SOLUTION(T, ad, m, t)。其中的第 1~2 行是递归头,表示递归结束的条件。第 3~10 行嵌套的 **if** 结构根据 $ad[i, j]$的取值,决定文本 T 中构成模式 P 的最佳近似匹配的字符(见图 11-14)。

图 11-14 展示了对 T = "james, peirce, dewey", P = pierce 运行 APPROXIMATE-MATCHER(P, T)后所得表格 ad,运行算法 SOLUTION(T,ad,m,t)的过程。在此例中,APPROXIMATE-MATCHER 算得的数表 ad 中最后一行 $ad[6, 1..18]$的最小值 $ad[m, t]$为$ad[6, 12]$ (对应的格子带有浅色阴影),于是,调用 SOLUTION(T, ad, 6, 12)。由于 $ad[6, 12]=2= ad[5, 11]$(浅色阴影格子),故递归调用 SOLUTION(T, ad, 5, 11)。由于 $ad[5, 11]=2= ad[4, 10]$ (浅色阴影格子),递归调用 SOLUTION(T, ad, 4, 10)。由于 $ad[4, 10]=2=ad[3, 9]+1$(浅色阴影格子),递归调用 SOLUTION(T, ad, 3, 9)。同样 $ad[3, 9]=2=ad[2, 8]+1$(浅色阴影格子),递归调用 SOLUTION(T, ad, 2, 8)。而 $ad[2, 8]=1= ad[1, 7]+1$(浅色阴影格子),递归调用 SOLUTION(T, ad, 1, 7)。最后,$ad[1, 7]=0=ad[0, 6]$(深色阴影格子),到达递归头($i=0$)。逐级返回前打印对应的 $T[j]$ 得到 P=pierce 在 T 中的最佳近似匹配为 $T[6..12]$=peirce。

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
			j	a	m	e	s	,	p	e	i	r	c	e	,	d	e	w	e	y
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	p	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	i	2	2	2	2	2	2	2	1	1	1	2	2	2	2	2	2	2	2	2
3	e	3	3	3	3	2	3	3	2	1	2	2	3	2	3	3	2	3	2	3
4	r	4	4	4	4	3	3	4	3	2	2	2	3	3	3	4	3	3	3	3
5	c	5	5	5	5	4	4	4	4	3	3	3	2	3	4	4	4	4	4	4
6	e	6	6	6	6	5	5	5	5	4	4	4	3	2	3	4	4	5	4	5

图 11-14 运行算法 SOLUTION(T, ad, m, t)的过程

由于调用输出最佳匹配的运行时间取决于递归次数,而每次递归 i、j 至少有一个自减 1,故递归次数为 $O(n+m)$。总之,可以在 $\Theta(mn)$时间内完成计算文本 $T[1..n]$中模式 $P[1..m]$的最佳近似匹配。

11.3.3　程序实现

细心的读者会发现，计算两个单词 P、W 的最小编辑距离的算法 11-9 和计算在文本 T 中查找模式 P 的最佳近似匹配的算法 11-11 几乎是一样的。区别在于前者是计算 $P[1..i]$和 $W[1..j]$的最小编辑距离 $d[i, j]$，而后者是计算 $P[1..i]$与 $T[1..j]$的所有后缀的最小编辑距离 $ad[i, j]$。体现在算法的伪代码上，仅第 4 行对 $d[0, j]$和对 $ad[0, j]$的赋值不同，前者赋值为 j，而后者赋值为 0。这是因为对前者而言，$d[0, j]$表示空串 ε 转换为 $W[1..j]$需要添加 j 个字符；而对于后者，$ad[0, j]$表示空串 ε 在 $T[1..j]$的所有后缀中，仅与空串 ε 最匹配。这样，此处仅罗列出算法 11-11 的实现代码，读者可打开源文件 utility/textmatch.c 查看本章其他算法（不包括各个应用问题中所列举的算法）的实现代码。

```
1 int *approximateMatch(char *p, char *t){
2     int m=strlen(p), n=strlen(t);
3     int *ad=(int *)malloc((m+1)*(n+1)*sizeof(int));
4     for(int j=0; j<=n; j++)
5         ad[j]=0;
6     for(int i=1; i<=m; i++)
7         ad[i*(n+1)]=i;
8     for (int i=1; i<=m; i++) {
9         for (int j=1; j<=n; j++) {
10            int a=min(ad[(i-1)*(n+1)+j],ad[i*(n+1)+j-1])+1,
11            b=ad[(i-1)*(n+1)+j-1];
12            if(p[i-1]!=t[j-1])b++;
13            ad[i*(n+1)+j]=min(a, b);
14        }
15    }
16    return ad;
17 }
```

程序 11-6　实现算法 11-11 的 C 函数

程序 11-6 的代码结构与算法 11-11 的伪代码结构是一样的。此处仅向读者指出，与本书之前的做法相同，用一维数组按行优先方式存储二维数组 ad[0..m, 0..n]。第 3 行为此数组分配存储空间。利用本函数返回的数组 ad，在该数组的最后一行找到最小值 ad[m, t]，调用下列实现算法11-12 的函数 solution(t, ad, m, t)，即可展示文本 t 中模式 p 的最佳匹配。

```
void solution(char *t, int *ad, int i, int j){
    if(i==0)
        return;
    int n=strlen(t);
    if((ad[i*(n+1)+j]==ad[(i-1)*(n+1)+j-1])||(ad[i*(n+1)+j]==ad[(i-1)*
    (n+1)+j-1]+1)){
```

```
            solution(t, ad, i-1, j-1);
            putchar(t[j-1]);
    }else if(ad[i*(n+1)+j]==ad[i*(n+1)+j-1]+1){
            solution(t, ad, i, j-1);
            putchar(t[j-1]);
      }else
            solution(t, ad, i-1, j);
}
```

程序 11-7　实现算法 11-12 的 C 函数

程序 11-7 的代码结构与算法 11-12 的为代码结构也是一样的。注意,由于是按行优先方式将二维数组 ad[0..m, 0..n]存储于一维数组 ad[0..n * m+1],因此访问 ad[i, j]的形式是 ad[i * (n+1)+j]。

11.3.4　应用

String Matching

Description

It's easy to tell if two words are identical—just check the letters. But how do you tell if two words are almost identical? And how close is "almost"?

There are lots of techniques for approximate word matching. One is to determine the best substring match, which is the number of common letters when the words are compared letter-by letter.

The key to this approach is that the words can overlap in any way. For example, consider the words CAPILLARY and MARSUPIAL. One way to compare them is to overlay them:

CAPILLARY
MARSUPIAL

There is only one common letter (A). Better is the following overlay:

CAPILLARY
MARSUPIAL

with two common letters (A and R), but the best is:

CAPILLARY
MARSUPIAL

Which has three common letters (P, I and L).

The approximation measure appx($word_1$, $word_2$) for two words is given by:

common letters * 2

——————————————————————————————

length(word1)+length(word2)

Thus, for this example, appx(CAPILLARY, MARSUPIAL)=6/(9+9)=1/3. Ob-

viously, for any word W appx(W, W)=1, which is a nice property, while words with no common letters have an appx value of 0.

Input

The input for your program will be a series of words, two per line, until the end-of-file flag of-1.

Using the above technique, you are to calculate appx() for the pair of words on the line and print the result.

The words will all be uppercase.

Output

Print the value for appx() for each pair as a reduced fraction. Fractions reducing to zero or one should have no denominator.

Sample Input

CAR CART
TURKEY CHICKEN
MONEY POVERTY
ROUGH PESKY
A A
−1

Sample Output

appx(CAR,CART)=6/7
appx(TURKEY,CHICKEN)=4/13
appx(MONEY,POVERTY)=1/3
appx(ROUGH,PESKY)=0
appx(A,A)=1

1. 问题描述与分析

寻求两个单词的近似匹配程度有很多方法,本问题就给出了一个方法:

计算 W_1 和 W_2 在任何交错形式下,对应字符相同的个数,取其中最大者的 2 倍除以两个单词长度的和作为它们的近似匹配度。

题中给出两个样例单词:W_1=CAPILLARY,W_2=MARSUPIAL。两者的几种交错形式如图 11-15 所示。

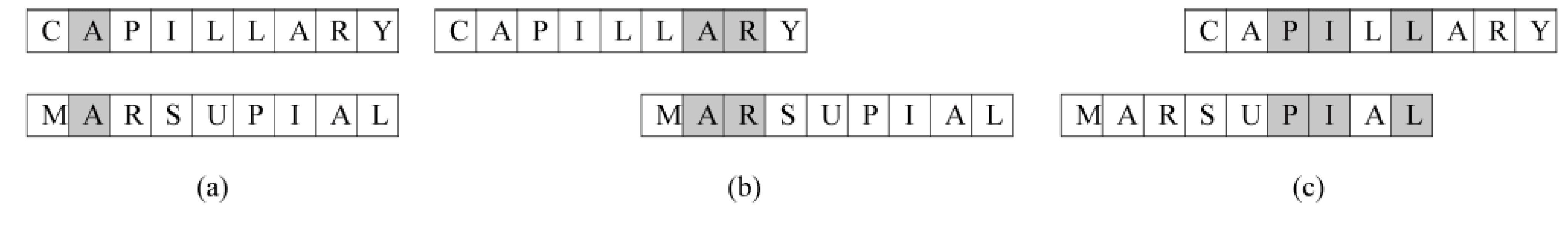


图 11-15　在不同交错形式下对应字符相同数的计算

图 11-15 反映了 W_1=CAPILLARY,W_2=MARSUPIAL 在不同交错形式下对应字符相同数的计算。图 11-5(a)中展示的形式下,只有一个对应相同字符 A(表示成带有阴影的

格子)。图 11-15(b)中有两个对应相同的字符 A 和 R。图 11-15(c)中有 3 个对应相同的字符 P、I 和 L。注意,W_1 和 W_2 的交错形式可以通过固定一个而向左移动另一个叠交形成。例如图 11-15(b)可视为固定 W_1,移动 W_2 形成;而图 11-15(c)可视为固定 W_2,移动 W_1 形成。

2. 算法描述

要解决本题,首先需要有一个计算给定的两个单词在任意交错形式下最大的对应相同字符数。下列算法给出了这个计算过程。

```
COMMON-LETTERS(word₁, word₂)
m←length[word₁], n←length[word₂]
max←0
for i←1 to m ▷固定 word₁,移动 word₂
    do count←0 ▷对应字符相同数计数器
       for j←1 to min{n, m−i+1}
          do if word₁[i+j−1]=word₂[j]
                then count←count+1
       if count>max ▷跟踪对应字符相同数的最大者
          then max←count
for i←1 to n ▷固定 word₂,移动 word₁
    do count←0
       for j←1 to min{m, n−i+1}
          do if word₂[i+j−1]=word₁[j]
                then count←count+1
       if count>max
          then max←count
return max
```

算法 11-13 计算两个单词任何交错形式下最大的对应相同字符数的过程

注意,$\min\{n, m-i+1\}$表示固定 W_1,移动 W_2 时,交错部分的长度。类似地,$\min\{m, n-i+1\}$表示固定 $word_2$,移动 $word_1$ 时,交错部分的长度。显然,过程 COMMON-LETTERS 的运行时间为 $\Theta(mn)$。利用 COMMON-LETTERS($word_1$, $word_2$),按照下列公式

$$\frac{\text{COMMON-LETTERS}(word_1, word_2)}{length(word_1)+length(word_2)}$$

计算出 $word_1$ 和 $word_2$ 的近似程度算法过程如下。

```
APPX(word₁, word₂)
m←length[word₁], n←length[word₂]
max←COMMON-LETTERS(word₁, word₂)
return 2 * max/(m+n)
```

算法 11-14 计算两个单词近似匹配程度的过程

算法 APPX 的运行时间也是 $\Theta(mn)$。

3. 程序实现

实现算法 11-13 的 C 函数代码如下。其代码结构与算法 11-3 的为代码结构是一致的，读者可对照研读。

```
int commonLetters(char * word1, char * word2){
    int max=0;
    int m=strlen(word1), n=strlen(word2);
    for(int i=0; i<m; i++){
        int count=0, l=n<m-i? n:m-i;        /* l 表示交错部分长度 min{n, m-i} */
        for(int j=0; j<l; j++)
            if(word1[i+j]==word2[j])
                count++;
        if(count>max)
            max=count;
    }
    for(int i=0; i<n; i++){
        int count=0, l=m<n-i? m:n-i;        /* l 表示交错部分长度 min{m, n-i} */
        for(int j=0; j<l; j++)
            if(word2[i+j]==word1[j])
                count++;
        if(count>max)
            max=count;
    }
    return max;
}
```

程序 11-8　实现算法 11-13 的 C 函数

实现算法 11-14 的 C 函数定义如下。

```
1 Rational appx(char * word1, char * word2){
2     int numerator=commonLetters(word1, word2),
3     denominator=strlen(word1)+strlen(word2);
4     Rational r={2 * numerator, denominator, 0};
5     normalize(&r);
6     return r;
7}
```

程序 11-9　实现算法 11-14 的 C 函数

程序 11-9 的代码结构与算法 11-14 的伪代码结构也是一样的。要注意的是，利用 4.5.3 节中定义的有理数结构体类型 Rational 表示两个单词近似程度，以满足题目对输出格式的要求。其中第 5 行调用的函数 normalize 负责对有理数进行约分操作。解决该问题完整的程序代码存储于源文件 chap11/ String Matching/ StringMatching. c。

第12章　代码实验

本书的最重要目标之一是和读者一起体验算法理论对程设计时的指导作用，以及程序设计实践反过来对加深对算法思想的理解的过程。本书的前10章在讨论经典问题算法的同时用C语言将其一一实现为通用的功能函数，形成了一个颇具规模的函数库。这些函数的源代码通过反复调试，存储于代码资源文件夹laboratory中，提供给读者研读、实验。根据自己在实践中的需要加以扩展，成为读者自己的解决实际应用问题的函数库。作为本书的结束，本章向读者介绍如何使用这些代码的基本要点。

12.1　头文件清单

首先将书中实现的函数原型声明所在的头文件罗列出来，供读者检索、查阅。书中的这些实现的基础数据类型及基础函数分门别类地存储于utility、datastructure、algebra、geometry、numbertheory、btrack、dprog、greedy和graph 9个文件夹内。本节按它们在书中出现的先后顺序罗列出各文件夹中的头文件及其内容。

12.1.1　基本应用类函数

1. 串输入输出流头文件strstream.h

```
typedef struct{
    char * begin;                                        /* 输入流首地址 */
    char * current;                                      /* 输入流当前读取位置 */
}StrInputStream;                                         /* 串输入流类型 */
void initStrInputStream(StrInputStream * ,char * );      /* 初始化输入流 */
int sisEof(StrInputStream * );                           /* 检测输入流是否结束 */
int readInt(StrInputStream * ,int * );                   /* 从输入流中读取整数 */
int readLong(StrInputStream * ,long * );                 /* 从输入流中读取整数 */
int readLongLong(StrInputStream * ,long long * );        /* 从输入流中读取整数 */
int readDouble(StrInputStream * ,double * );             /* 从输入流中读取双精度浮点数 */
int readChar(StrInputStream * ,char * );                 /* 从输入流中读取字符 */
int readString(StrInputStream * ,char * );               /* 从输入流中读取字符串 */
void sisRewind(StrInputStream * );                       /* 输入流还原初始设置 */
typedef struct{
    char * begin;                                        /* 输出流首地址 */
    char * current;                                      /* 输出流当前写入位置 */
    int length;                                          /* 输出流长度 */
}StrOutputStream;                                        /* 串输出流类型 */
```

```
void initStrOutputStream(StrOutputStream * ,int);         /* 初始化输出流 */
void freeStrOutputStream(StrOutputStream * );
int sosFull(StrOutputStream * );                          /* 检测输出流满 */
int writeInt(StrOutputStream * ,int);                     /* 向输出流写入整数 */
int writeDouble(StrOutputStream * ,double);               /* 向输出流写入浮点数 */
int writeChar(StrOutputStream * ,char);                   /* 向输出流写入字符 */
int writeString(StrOutputStream * ,char * );              /* 向输出流写入字符串 */
void sosRewind(StrOutputStream * );                       /* 输出流还原初始设置 */
StrOutputStream ssout;                                    /* 全局串输出流对象 */
void putInt(int * );                                      /* 向 ssout 写入整数 */
void putChar(char * );                                    /* 向 ssout 写入字符 */
void putDouble(double * );                                /* 向 ssout 写入浮点数 */
void putString(char * );                                  /* 向 ssout 写入字符串 */
```

2. 基本数据比较头文件 compare. h

```
int intGreater(int const * x,int const * y);              /* 整型数据大于比较 */
int intLess(int const * x,int const * y);                 /* 整型数据小于比较 */
int charGreater(char const * x,char const * y);           /* 字符数据大于比较 */
int charLess(char const * x,char const * y);              /* 字符数据小于比较 */
int strGreater(char const ** x,char const ** y);          /* 字符串数据大于比较 */
int strLess(char const ** x,char const ** y);             /* 字符串数据小于比较 */
int floatGreater(float const * x,float const * y);        /* 单精度浮点数据大于比较 */
int floatLess(float const * x,float const * y);           /* 单精度浮点数据小于比较 */
int doubleGreater(double const * x,double const * y);     /* 双精度浮点数据大于比较 */
int doubleLess(double const * x,double const * y);        /* 双精度浮点数据小于比较 */
int unsignedGreater(unsigned const * x,unsigned const * y); /* 无符号整数大于比较 */
int unsigned_long_longGreater(unsigned long long const * x,
unsigned long long const * y);                            /* 无符号长整数大于比较 */
int dblLess(double const ** x,double const ** y);        /* 双重指针指引的双精度数据大于比较 */
```

3. 随机数发生器头文件 randomn. h

```
int randomNumber(int p, int q);                    /* 返回 p~q 之间的一个随机整型数 */
unsigned long long RangedRandom( unsigned long long range_min, unsigned long long range_max);
                                                   /* 返回 min~max 之间的一个随机长整型数 */
```

4. 交换变量数据头文件 swap. h

```
void swap(void * x,void * y,int size);                    /* 交换 x、y 指向的变量的值 */
```

5. 数组合并头文件 merge. h

```
void merge(void * a, int size, int p, int q, int r,int( * comp)(void * ,void * ));
                                /* 有序数组 a[p..q]、a[q+1..r]合并为有序数组 a[p..r] */
```

6. 链表合并头文件 listmerge. h

```
void listMerge(LinkedList * A,          /* 合并链表 A 中的有序子序列 p~q、q~r 为有序序列 */
ListNode * p, ListNode * q, ListNode * r);
```

7. 数组归并排序头文件 mergesort.h

```
void mergeSort(void *a,int size,int p,int r,int(*comp)(void *,void *));
                                          /*对数组a[p..r]做归并排序*/
```

8. 链表归并排序头文件 listmergesort.h

```
void listMergeSort(LinkedList *A, ListNode *p, ListNode *r);    /*链表A的归并排序*/
```

9. 数组划分头文件 partition.h

```
long partition(void *a,int size,int p,int r,int(*comp)(void *,void *));
                                   /*对数组a[p..r]划分,返回分界点下标*/
long randmizedPartition(void *a,int size,long p,long r,int(*comp)(void *,void *));
                                   /*对数组a[p..r]划分的随机版本*/
```

10. 链表划分头文件 listpartition.h

```
ListNode *listPartition(LinkedList *A, ListNode *p,ListNode *r);      /*链表划分*/
ListNode *rndListPartition(LinkedList *A, ListNode *p,ListNode *r);
                                                   /*链表划分的随机版本*/
```

11. 数组快速排序头文件 quicksort.h

```
void quickSort(void *a,int size,int p,int r,int(*comp)(void *,void *));
                                          /*对数组a[p..r]做快速排序*/
```

12. 链表快速排序头文件 listquicksort.h

```
void listQuickSort(LinkedList *A, ListNode *p, ListNode *r);          /*链表的快速排序*/
```

13. 数组选择头文件 select.h

```
void *select(void *a, int size, int p, int r,int i,int(*comp)(void *,void *));
                                       /*在数组a[p..r]中找第i小的元素*/
```

14. 链表选择头文件 listselect.h

```
ListNode *listSelect(LinkedList *a, ListNode *p, ListNode *r,int i);
                                     /*在链表a中找q~r之间的第i小元素*/
```

15. 计数排序头文件 countsort.h

```
void countSort(unsigned *a,int n);                 /*计数排序*/
```

16. 数组最大之最小值头文件 most.h

```
int most(void *a, int n, int size, int(*comp)(void *,void *));
                             /*计算数组a[0..n-1]中的最大/最小值的下标*/
```

17. 基本数据输出头文件 output.h

```
void intOutput(void *x);                    /*输出x指向的整型数据*/
void charOutput(void *x);                   /*输出x指向的字符型数据*/
```

```
void doubleOutput(void * x);                    /* 输出 x 指向的双精度浮点型数据 */
void floatOutput(void * x);                     /* 输出 x 指向的单精度浮点型数据 */
void stringOutput(void * x);                    /* 输出 x 指向的字符串型数据 */
```

18. 线性查找头文件 search.h

```
int linearSearch(void * a,int size, int n, void * x,int( * comp)(void * ,void * ));
                                                /* 在数组 a[0..n-1]中线性查找值 * x * /
```

19. 堆操作头文件 heap.h

```
int left(int i);                                /* 计算 a[i]的左孩子下标 */
int right(int i);                               /* 计算 a[i]的右孩子下标 */
int parent(int i);                              /* 计算 a[i]的父亲下标 */
void heapify(void * a,int size,int i,int heapSize,int( * comp)(void const * ,void const * ));
                                                /* 维护 a[i]的堆性质 */
void buildHeap(void * a,int size, int length,int( * comp)(void const * ,void const * ));
                                                /* 创建堆 */
```

12.1.2　数据结构类

1. 序偶头文件 pair.h

```
typedef struct {                                /* 用来存储两个指针的结构体 */
    void * first;
    void * second;
} pair;
pair make_pair(void * f, void * d);             /* 创建 pair 对象 */
```

2. 链表头文件 list.h

```
typedef struct ListNode{                        /* 结点类型结构体 */
    void * key;                                 /* 关键值指针 */
    struct ListNode * prev;                     /* 前驱结点指针 */
    struct ListNode * next;                     /* 后继结点指针 */
} ListNode;
ListNode * createListNode(void * d, int size);  /* 创建结点 */
void clrListNode(ListNode * x, void( * proc)(void * ));  /* 清理结点 */
typedef struct {                                /* 链表类型结构体 */
    size_t eleSize;                             /* 元素数据存储宽度 */
    int ( * comp)(void * , void * );            /* 元素数据比较规则 */
    ListNode * nil;                             /* 头指针 */
    int n;                                      /* 元素个数 */
}LinkedList;
LinkedList * createList(unsigned long size,int( * comp)(void * ,void * ));  /* 创建新链表 */
void clrList(LinkedList * L, void( * proc)(void * ));   /* 清理链表 */
int listEmpty(LinkedList * );                   /* 检测链表是否为空 */
```

```
ListNode * listSearch(LinkedList * L, void * e);                /* 在链表中查找 */
void listInsert(LinkedList * L, ListNode * a, void * k); /* 在 L 的结点 a 前插入 key 为 k 的结点 */
void listPushFront(LinkedList * L, void * k);                    /* 在表首插入 */
void listPushBack(LinkedList * L, void * k);                     /* 在表尾插入 */
void listDelete(LinkedList * L, ListNode * e);                   /* 在链表中删除结点 e */
void listTraverse(LinkedList * L, void( * proc)(void * ));       /* 遍历链表 */
ListNode * advance(ListNode * i,int d);
int distance(ListNode * p, ListNode * r);
LinkedList * listClone(LinkedList * );
```

3. 栈头文件 stack.h

```
typedef struct{                                                  /* 栈类型 */
    LinkedList * L;                                              /* 链表属性 */
    ListNode * top;                                              /* 栈顶属性 */
}Stack;
Stack * createStack(unsigned long size);                         /* 创建空栈 */
void clrStack(Stack * S, void( * proc)(void * ));
int stackEmpty(Stack * S);                                       /* 检测栈空 */
void push(Stack * S,void * k);                                   /* 压栈操作 */
ListNode * pop(Stack * S);                                       /* 弹出操作 */
```

4. 队列头文件 queue.h

```
typedef struct{                                                  /* 队列类型 */
    LinkedList * L;                                              /* 链表属性 */
    ListNode * head, * tail;                                     /* 队首队尾属性 */
}Queue;
Queue * createQueue(unsigned long size);                         /* 创建空队列 */
void clrQueue(Queue * Q, void( * proc)(void * ));                /* 清理队列空间 */
int queueEmpty(Queue * Q);                                       /* 检测队列空 */
void enQueue(Queue * Q,void * k);                                /* 入队操作 */
ListNode * deQueue(Queue * Q);                                   /* 出队操作 */
```

5. 优先队列头文件 pqueue.h

```
typedef struct {
    void * heap;
    int eleSize;
    int length;
    int heapSize;
    int( * compare)(void const * ,void const * );
}PQueue;
PQueue * initPQueue(int size,int n,int( * comp)(void const * ,void const * ));
void pQueueClr(PQueue * q);
int empty(PQueue * q);
void enQueue(PQueue * q,void * e);
void * top(PQueue * q);
```

```
void *deQueue(PQueue *q);
void fix(PQueue *q);
```

6. 二叉树头文件 binarytree.h

```
typedef struct node{
    struct node *p;                                    /*父结点指针*/
    struct node *left;                                 /*左孩子结点指针*/
    struct node *right;                                /*右孩子结点指针*/
    void *key;                                         /*数据域指针*/
}BTreeNode;                                            /*二叉树结点类型*/
BTreeNode *creatBTreeNode(void *key, int size);        /*创建结点*/
void clrBTreeNode(BTreeNode *r,void(*proc)(void*));    /*清理二叉树结点*/
BTreeNode *creatBTree(void *k, int size, BTreeNode *l, BTreeNode *r);  /*创建树*/
void clrBTree(BTreeNode *r,void(*proc)(void*));        /*清理二叉树*/
void inorderTreeWalk(BTreeNode *r, void (*proc)(void *));   /*中序遍历以r为根的二叉树*/
void preorderTreeWalk(BTreeNode *r, void (*proc)(void *));  /*前序遍历*/
void postorderTreeWalk(BTreeNode *r, void (*proc)(void *)); /*后序遍历*/
```

7. 红-黑树头文件 redblacktree.h

```
typedef enum {RED, BLACK} Color;
typedef struct node{                                   /*二叉树结点类型*/
    struct node *p;                                    /*父结点指针*/
    struct node *left;                                 /*左孩子结点指针*/
    struct node *right;                                /*右孩子结点指针*/
    Color color;
    void *key;                                         /*数据域指针*/
}RBNode;
RBNode *creatRBNode(void *key, int size);              /*创建结点*/
void clrRBNode(RBNode *r,void(*proc)(void*));          /*清理二叉树结点*/
typedef struct{
    unsigned nodeSize;
    int (*comp)(void*, void*);
    RBNode *root;
    RBNode *nil;
}RBTree;
RBTree *creatRBTree(int size, int(*comp)(void *,void *));   /*创建空树*/
void clrRBTree(RBTree *t,void(*proc)(void*));          /*清理二叉树*/
void inorderRBWalk(RBTree *t, void (*proc)(void *)) ;  /*中序遍历以r为根的二叉树*/
/*二叉搜索树的操作*/
RBNode *rbMini(RBTree *t);                             /*计算最小值*/
RBNode *rbMax(RBTree *t);                              /*计算最大值*/
RBNode *treeSuccessor(RBNode *r);                      /*计算r的后继*/
RBNode *treePredecessor(RBNode *r);                    /*计算r的前驱*/
RBNode *rbSearch(RBTree *t,void *k);                   /*在搜索树中查找*/
RBNode *rbInsert(RBTree *t,void *key);                 /*向搜索树插入元素*/
```

```
RBNode * rbDelete(RBTree * t,RBNode * x);
```

8. 散列表头文件 hstb.h

```
typedef struct {                                              /* hash 表结构 */
    LinkedList ** table;                                      /* 槽位数组 */
    size_t m;                                                 /* 槽位数 */
    size_t n;                                                 /* 元素个数 */
} Hash_Table;
Hash_Table * createTable(int c);                              /* 创建具有 c 个槽位的 hash 表 */
void clrTable(Hash_Table * t);
int tbIsEmpty(Hash_Table * t);                                /* 检测表空 */
int hashInsert(Hash_Table * t, unsigned long long key);       /* 向表 t 中插入元素 key */
int inHashTable(Hash_Table * t, unsigned long long key);      /* 在表 t 中查找关键值 key */
int hashDelete(Hash_Table * t, int unsigned long long ele);   /* 在表 t 中删除元素 ele */
```

9. 大整数散列表头文件 hstb1.h

```
typedef struct {                                              /* hash 表结构 */
    LinkedList ** table;                                      /* 槽位数组 */
    BigInt m;                                                 /* 槽位数 */
    size_t n;                                                 /* 元素个数 */
} HashTable;
HashTable * CreateTable(long c);                              /* 创建具有 c 个槽位的 hash 表 */
void ClrTable(HashTable * t);
int TbIsEmpty(HashTable * t);                                 /* 检测表空 */
int HashInsert(HashTable * t, BigInt key);                    /* 向表 t 中插入元素 key */
int InHashTable(HashTable * t, BigInt key);                   /* 在表 t 中查找关键值 key */
int HashDelete(HashTable * t, BigInt key);                    /* 在表 t 中删除元素 key */
```

12.1.3 代数记算类函数

1. 矩阵头文件 matrix.h

```
typedef struct{                                               /* 矩阵类型定义 */
    double * tab;                                             /* 二维数表 */
    int row,col;                                              /* 行数、列数 */
}Matrix;
Matrix newMatrix(int, int);                                   /* 生成矩阵 */
Matrix newMatrixByArry(double * , int, int);                  /* 用数组数据生成矩阵 */
void clrMatrix(Matrix);                                       /* 清理矩阵空间 */
void printMatrix(Matrix);                                     /* 输出矩阵 */
Matrix matrixAdd(Matrix, Matrix);                             /* 矩阵相加 */
Matrix matrixDiff(Matrix, Matrix);                            /* 矩阵相减 */
Matrix matrixMultiply(Matrix, Matrix);                        /* 矩阵相乘 */
Matrix transform(Matrix);                                     /* 矩阵转置 */
void swapRows(Matrix a, int i, int j);                        /* 交换两行 */
```

```
typedef struct{
    Rational * tab;                                   /* 二维数表 */
    int row,col;                                      /* 行数、列数 */
}rMatrix;
rMatrix rnewMatrix(int, int);                         /* 生成矩阵 */
rMatrix rnewMatrixByArry(Rational *, int, int);       /* 用数组数据生成矩阵 */
void rclrMatrix(rMatrix);                             /* 清理矩阵空间 */
void rprintMatrix(rMatrix a);
void rswapRows(rMatrix a, int i, int j);              /* 交换两行 */
```

2. LUP 分解头文件 lup.h

```
Matrix lupDecomposition(Matrix a, int * pi);          /* 对矩阵 a 做 lup 分解 */
double * lupSolve(Matrix lu, int * pi, double * b);   /* 基于 lup 分解解线性方程组 */
Matrix matrixInverse(Matrix a);                       /* 计算矩阵 a 的逆矩阵 */
rMatrix rlupDecomposition(rMatrix a, int * pi);       /* 有理数矩阵 a 的 lup 分解 */
```

3. 复数头文件 complex.h

```
typedef struct{
    double real;                                      /* 实部 */
    double magic;                                     /* 虚部 */
}Complex;                                             /* 复数类型 */
void printComplex(Complex);                           /* 复数的输出 */
int isZero(Complex);                                  /* 检测是否为 0 */
Complex compSum(Complex,Complex);                     /* 复数相加 */
Complex compDiff(Complex,Complex);                    /* 复数相减 */
Complex compProd(Complex,Complex);                    /* 复数相乘 */
Complex compQuot(Complex,Complex);                    /* 复数相除 */
```

4. 复系数多项式头文件 polynomial.h

```
typedef struct{
    Complex * coeff;                                  /* 系数表 */
    int degree;                                       /* 次数 */
}Polynomial;                                          /* 复系数多项式类型 */
Polynomial newPoly(int n);                            /* 生成多项式 */
Polynomial newPolyByArry(double *, int n);            /* 用数组数据生成多项式 */
void clrPoly(Polynomial);                             /* 清理多项式空间 */
void printPoly(Polynomial);                           /* 输出多项式 */
Polynomial polySum(Polynomial,Polynomial);            /* 多项式的和 */
Polynomial polyProd(Polynomial f, Polynomial g);
double horner(Polynomial, double);                    /* 霍纳法计算多项式的值 */
```

5. FFT 头文件 fft.h

```
Complex * fft(Complex * a, int n, int inv);       /* 向量 a 的 fft 变换 */
Complex * fftInverse(Complex * a, int n);         /* 向量 a 的 fft 逆变换 */
```

6. 实系数多项式头文件 poly.h

```
typedef struct{
    double * coeff;                                    /* 系数向量 */
    int degree;                                        /* 次数 */
}Poly;                                                 /* 实系数多项式类型 */
Poly newPoly(int n);                                   /* 创建 n 次多项式 */
Poly newPolyByArry(double * a, int n);                 /* 用实系数向量 a 创建 n 次多项式 */
void clrPoly(Poly);                                    /* 清理多项式存储空间 */
void printPoly(Poly);                                  /* 输出多项式 */
void polyAssign(Poly * ,Poly);                         /* 多项式赋值 */
Poly polySum(Poly,Poly);                               /* 多项式相加 */
Poly polyProd(Poly f, Poly g);                         /* 多项式相乘 */
double horner(Poly, double);                           /* 霍纳法求值 */
Poly derivative(Poly);                                 /* 多项式导数 */
void nomalPoly(Poly * );                               /* 规格化多项式 */
```

7. 有理数头文件 rational.h

```
typedef struct{
    unsigned numerator;                                /* 分子 */
    unsigned denominator;                              /* 分母 */
    int sign;                                          /* 符号 */
}Rational;                                             /* 有理数类型 */
void printRational(Rational);                          /* 输出有理数 */
int rationalIsZero(Rational);                          /* 检测有理数是否为 0 */
int valueCompare(Rational a, Rational b);              /* 有理数大小比较 */
void normaliz(Rational * );                            /* 规格化有理数 */
Rational rationalSum(Rational,Rational);               /* 有理数相加 */
Rational rationalDiff(Rational,Rational);              /* 有理数相减 */
Rational rationalProd(Rational,Rational);              /* 有理数相乘 */
Rational rationalQuot(Rational,Rational);              /* 有理数相除 */
```

12.1.4 计算几何类函数

1. 平面点头文件 point.h

```
typedef struct{                                        /* 平面点 */
    double x,y;                                        /* 横坐标与纵坐标 */
}Point;
double dist(Point * a, Point * b);                     /* 两点间距离 */
double crossProduct(Point * a, Point * b);             /* 叉积 */
Point sub(Point * a, Point * b);                       /* 两点向量差 */
int direction(Point * p0, Point * p1, Point * p2); /* 判断向量 p2p0 从向量 p1p0 绕 p0 的旋转方向 */
int inBox(Point * pi, Point * pj, Point * pk);  /* 检测 pk 是否落于以 pi、pj 为对角线的矩形内 */
int InBox(Point * pi, Point * pj, Point * pk);
```

```
double pabs(Point *a);
void normalize(Point *a);
int segmentsIntersect(Point *p1, Point *p2, Point *p3, Point *p4);
                                          /*检测线段 p1p2 和线段 p3p4 是否相交*/
double psudoPolarAngle(Point *a, Point *b);         /*计算伪极角*/
Point jointPoint(Point *a1, Point *b1, Point *a2, Point *b2);
                                          /*计算相交线段 a1b1、a2b2 的交点*/
void pointRotate(Point *p, double theta);
```

2. 平面线段头文件 segment.h

```
typedef struct{
    Point point;                            /*端点坐标*/
    int e;                                  /*端点标志*/
}EndPoint;                                  /*线段端点类型*/
int pointLess(EndPoint *a, EndPoint *b);    /*端点比较*/
typedef struct{
    EndPoint left;                          /*左端点*/
    EndPoint right;                         /*右端点*/
    double length;                          /*长度*/
    double tan;                             /*斜率*/
    double yOffset;                         /*所在直线在 y 轴的截距*/
}Segment;                                   /*线段类*/
Segment newSeg(Point *a, Point *b);         /*构造函数*/
EndPoint getLeft(Segment *a);               /*访问左端点*/
EndPoint getRight(Segment *a);              /*访问右端点*/
double getY(Segment *a, double x);          /*计算线段在 x 处的纵坐标*/
EndPoint getLeft(Segment *a);
EndPoint getRight(Segment *a);
int segComp(Segment *a, Segment *b);        /*线段比较*/
int anySegmentIntersect(Segment *S, int n);
```

3. 平面点集凸包头文件 convexhull.h

```
Point *grahamScan(Point *p, int n, int *m);
```

4. 平面点集最邻近点对头文件 closestpairpoints.h

```
double closestPairPoints(Point *x, Point *y, int n);    /*计算点集中最邻近点对距离*/
```

12.1.5 数论计算类函数

1. 大整数绝对值运算头文件 valuecalc.h

```
#define Base 1000000000
int valueCompare(LinkedList *a, LinkedList *b);
LinkedList *valueAdd(LinkedList *a, LinkedList *b);
LinkedList *valueSub(LinkedList *a, LinkedList *b);
```

```
void valueDecreas(LinkedList *a, LinkedList *b);
void valueReduplicate(LinkedList *a, long b);
LinkedList *multedByDigit(LinkedList *a, long x);
LinkedList *valueMulti(LinkedList *a, LinkedList *b);
void dividedByDigit(LinkedList *a, long x, long *r);
LinkedList *valueDivision(LinkedList *a, LinkedList *b, LinkedList *r);
```

2. 大整数运算头文件 bigint.h

```
typedef struct{
    LinkedList *value;
    int sign;
}BigInt;
BigInt newIntByint(long long);
BigInt newIntBystring(char *);
void clrBigInt(BigInt *);
void intAssign(BigInt *a, BigInt b);
void printInt(BigInt *);
char *toString(BigInt *a);
int compareInt(BigInt *, BigInt *);
int isZero(BigInt x);
int isOne(BigInt a);
BigInt negative(BigInt a);
BigInt sum(BigInt, BigInt);
void increase(BigInt *, unsigned);
BigInt diff(BigInt, BigInt);
void decrease(BigInt *, unsigned);
BigInt product(BigInt, BigInt);
BigInt quotient(BigInt, BigInt);
BigInt remainder(BigInt, BigInt);
```

3. 最大公约数头文件 gcd.h

```
typedef unsigned long long ul_int;
typedef long long l_int;
ul_int gcd(ul_int, ul_int);
ul_int egcd(ul_int a, ul_int b, l_int *x, l_int *y);
BigInt euclid(BigInt, BigInt);
BigInt extendedEuclid(BigInt, BigInt, BigInt *, BigInt *);
```

4. 模线性方程头文件 lequation.h

```
long long mod(long long a, long long m);
LinkedList *modularLinearEquationSolver(unsigned long long,    /*系数*/
                                        unsigned long long,    /*常量*/
                                        unsigned long long     /*模*/);
LinkedList *modularEquationSolver(BigInt, BigInt, BigInt);
```

5. 模幂头文件 exponent.h

```
unsigned long lg2(unsigned long long n);
unsigned long long modularExponentiation(unsigned long long a,
                                         unsigned long long b,
                                         unsigned long long modular);
BigInt modularExponent(BigInt a, BigInt b, BigInt m);
```

6. 素数检测头文件 primetest.h

```
int isPrime(unsigned long long);                    /* 常规判断素数 */
BigInt random(BigInt a);                            /* 随机大整数 */
int millerRabin(unsigned long long n, int s);       /* 随机算法判断素数 */
int MillerRabin(BigInt n, int s);                   /* 判断大素数 */
```

7. 因数分解头文件 factor.h

```
void factor(unsigned long n, RBTree * s);           /* 长整型数据的因数分解 */
void Factor(BigInt n, RBTree * s);                  /* 大整数的因数分解 */
```

12.1.6　回溯搜索类函数

1. 组合问题头文件 combineproblem.h

```
typedef struct{                                     /* 解向量分量取值集合类型 */
    LinkedList  * list;                             /* 链表 */
    ListNode  * current;                            /* list 当前结点指针 */
}ValueSet;
void clrSet(ValueSet * set, void( * proc)(void * ));   /* 清理取值集合存储空间 */
void clrOMG(ValueSet  ** OMG, int n, void( * proc)(void * ));  /* 清理解空间 */
typedef struct{                                     /* 合法解类型 */
    void  * x;                                      /* 解向量 */
    int k;                                          /* 解向量长度 */
}Solution;
void clrSolution(Solution * );                      /* 清理合法解存储空间 */
int isPartial(void * , int);                        /* 部分合法判断 */
int isComplete(void * , int);                       /* 完整合法解判断 */
LinkedList * backtrackiter(ValueSet  ** OMG, int n);   /* 组合问题回溯求解 */
LinkedList * subSetTree(int n);                     /* 子集树问题回溯求解 */
LinkedList * permuteTree(void  * origin, int size, int n);  /* 排列树问题回溯求解 */
```

2. m-着色问题头文件 mcolor.h

```
int  * G,                                           /* 图 G 的邻接矩阵 */
    n,                                              /* 图 G 的顶点个数 */
    m;                                              /* 颜色种数 */
ValueSet  * newSet();
ValueSet ** makeOMG();
```

12.1.7 动态规划类函数

1. 最长公共子序列头文件 lcs.h

```
int * lcsLength(void * x,void * y,int size, int m,int n,int(* comp)(void *,void *));
void printLcs(int * c,int n, void * x,void * y, int size, int i, int j,int(* comp)(void *,void *),
void(* prt)(void *));
```

2. 头文件 knapsack.h

```
int * knapsack(int * w,int * v,int n,int c);
int * buildSolution(int * m, int n,int * w, int c);
```

3. 任意点对最短路径头文件 floyd.h

```
pair floydWarshall(double * w,int n);
void printAllPairsShortestPath(int * pi,int n,int i,int j);
```

12.1.8 贪婪策略类函数

1. 相容活动头文件 actsel.h

```
typedef struct{
    double s;
    double f;
}Active;
int * greedyActivitySelector(Active * a, int n);
```

2. Huffman 编码头文件 huffman.h

```
typedef struct{
    int ch;
    int f;
    int parent;
    int * child;
    char code[16];
}Node;
typedef struct{
    Node * Nodes;
    int R;
    int N;
}HTree;
void clrHTree(HTree * t);
HTree Huffman(char * C, int * f, int n, int r);
```

3. 单源最短路径头文件 dijkstra.h

```
#define dblinfy DBL_MAX
pair dijkstra(double * w, int n, int s);
void printPath(int * pi,int s, int i);
```

4. 最小生成树头文件 prim.h

```
pair mstPrim(double * w,int n,int r);
```

12.1.9 图的搜索类函数

1. 图的邻接表头文件 graph.h

```
typedef struct{
    double weight;
    int index;
} vertex;
typedef struct Graph{
    LinkedList ** adj;
    int n;
} Graph;
typedef enum {WHITE,GRAY,BLACK} Color;
Graph * createGraph(double * a, int n);
Graph * zeroGraph(int n);
void graphClear(Graph * g);
Graph * transpose(Graph * g);
Graph * cloneGraph(Graph * const g);
void printGraph(Graph * g);
void addVertex(Graph * g);
void deleteVertex(Graph * g, int u);
void addEdge(Graph * g, int u, int v, double w);
void deleteEdge(Graph * g, int u, int v);
```

2. 深度优先搜索头文件 dfs.h

```
pair dfs(Graph * g);
```

3. 拓扑排序头文件 tpsort.h

```
pair toplogicalSort(Graph * g);
```

4. 强连通分支头文件 scc.h

```
LinkedList * strongConnectedComponents(Graph * g);
```

5. 关结点头文件 articpoint.h

```
LinkedList * articPoint(Graph * g, int s);
```

6. 深度优先搜索头文件 bfs.h

```
pair bfs(Graph * g,int s);
```

7. 最大流头文件 edmondskarp.h

```
double * edmondsKarp(double * c, int n, int s, int t);
```

12.1.10 文本搜索类函数

文本搜索头文件 textmatch.h

```
int find(char t, char p);                       /* 在文本串 t 中查找文本模式 p 第一个发生位置 */
int * manacher(char * t);                       /* 计算文本串 a 中以各元素为中心的回文子串长度 */
int optimalEditDistance(char * p, char * w);    /* 计算串 p 和 w 的编辑距离 */
int * approximateMatch(char * p, char * t);     /* 计算 p 在串 t 中的最佳近似匹配 */
void solution(char * t, int * ad, int i, int j); /* 显示 t 中 p 的最佳近似匹配 */
```

12.2 实验平台的搭建

12.2.1 集成开发环境的安装

本书中所有代码都是在 Microsoft 公司的 Windows 平台上,在 Visual Studio 2010 中实现的。Microsoft 公司为大学生提供了 Visual Studio 2010 的学习版。这是一款全免费的软件,读者可在 Microsoft 公司官方网站 http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express 上下载,下载后可安装。安装界面如图 12-1 所示。



图 12-1 Visual Studio 2010 学习版安装界面

12.2.2　实验项目的建立

在安装了 Visual Studio 2010 学习版的系统中启动 Visual Studio 2010，打开如图 12-2 所示的窗口。



图 12-2　Visual C++ 2010 启动后打开的窗口

单击“文件”→“新建”→“项目”命令，打开如图 12-3 所示的“新建项目”对话框。



图 12-3　“新建项目”对话框

在左窗格中点选“常规”，在中部窗格中点选“空项目”。

在“名称”栏中输入项目名，例如输入 laboratory。

在“位置”栏中选定项目的存储文件夹路径。

单击“确定”按钮，打开的 IDE 窗口如图 12-4 所示。

将文件夹 laboratory 中的所有子文件夹复制到硬盘上新建的项目文件夹 laboratory 的子文件夹 laboratory 中，如图 12-5 所示。

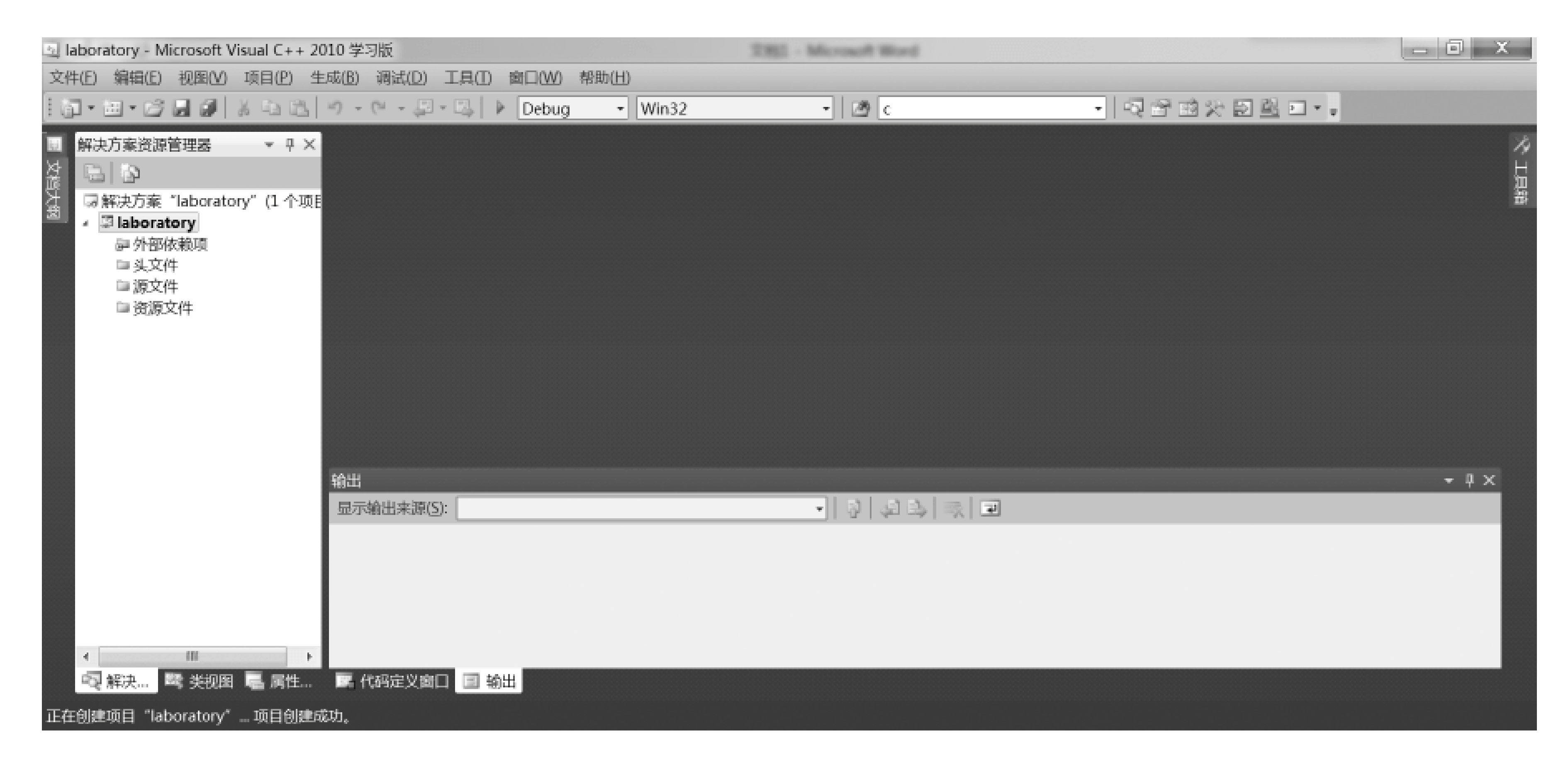


图 12-4　创建了项目后的 IDE 窗口

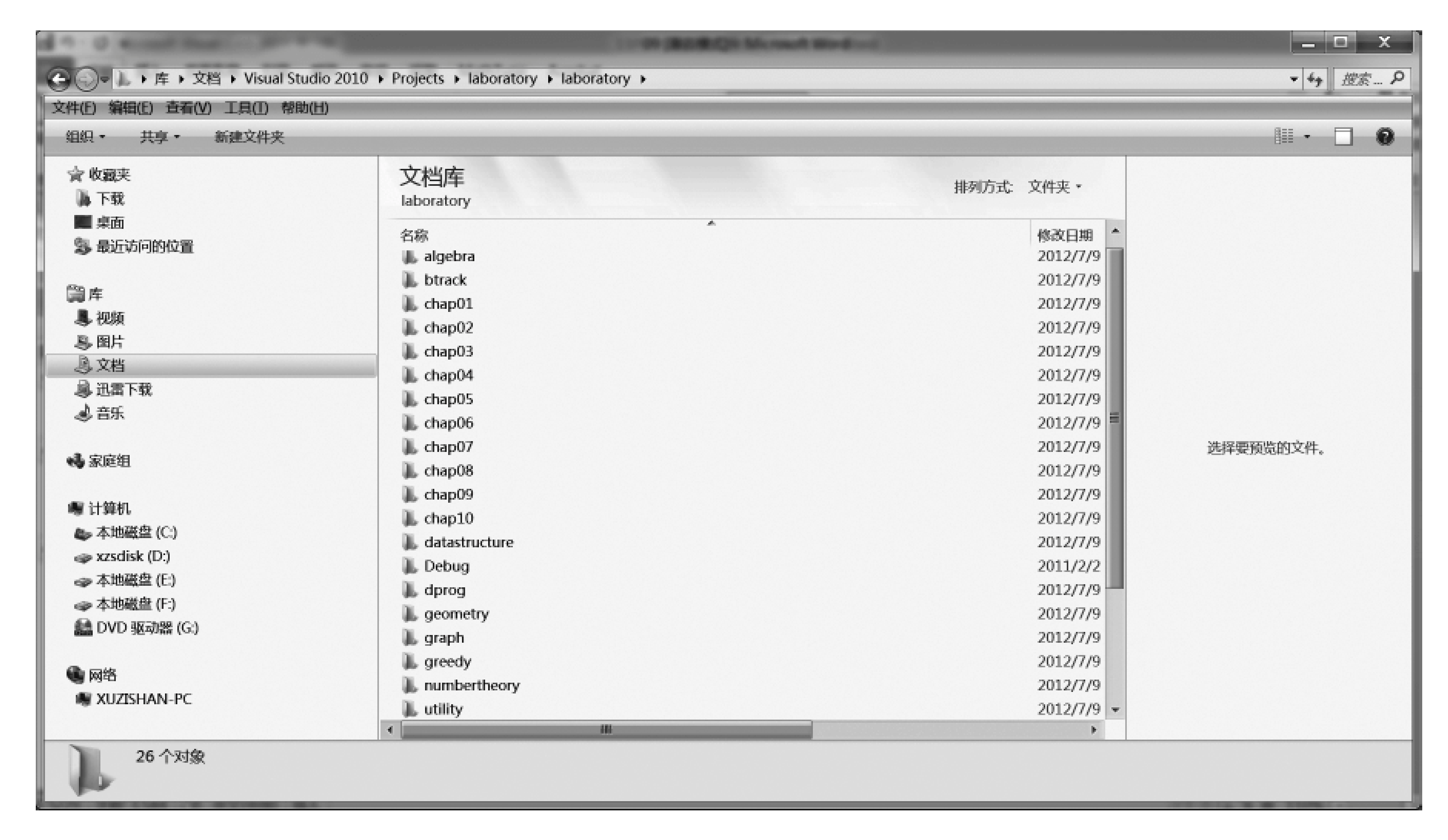


图 12-5　复制文件

12.3　应用问题程序的运行实例

要在 12.1 节中创建的实验项目内运行书中应用问题的程序,可按下列步骤进行。

12.3.1　加载程序文件

以运行本书第 1 章的程序 1-7 为例,说明如何加载运行该程序所需的源文件。

(1) 启动 Visual C++ 2010 学习版。

在 IDE 中打开在 12.1 节中建立的项目 laboratory,屏幕中出现图 12-4 中展示的窗口。

（2）右击项目右窗格中“源文件”图标。在弹出式菜单中单击“添加”→“现有项”，弹出“添加现有项“对话框，如图 12-6 所示。

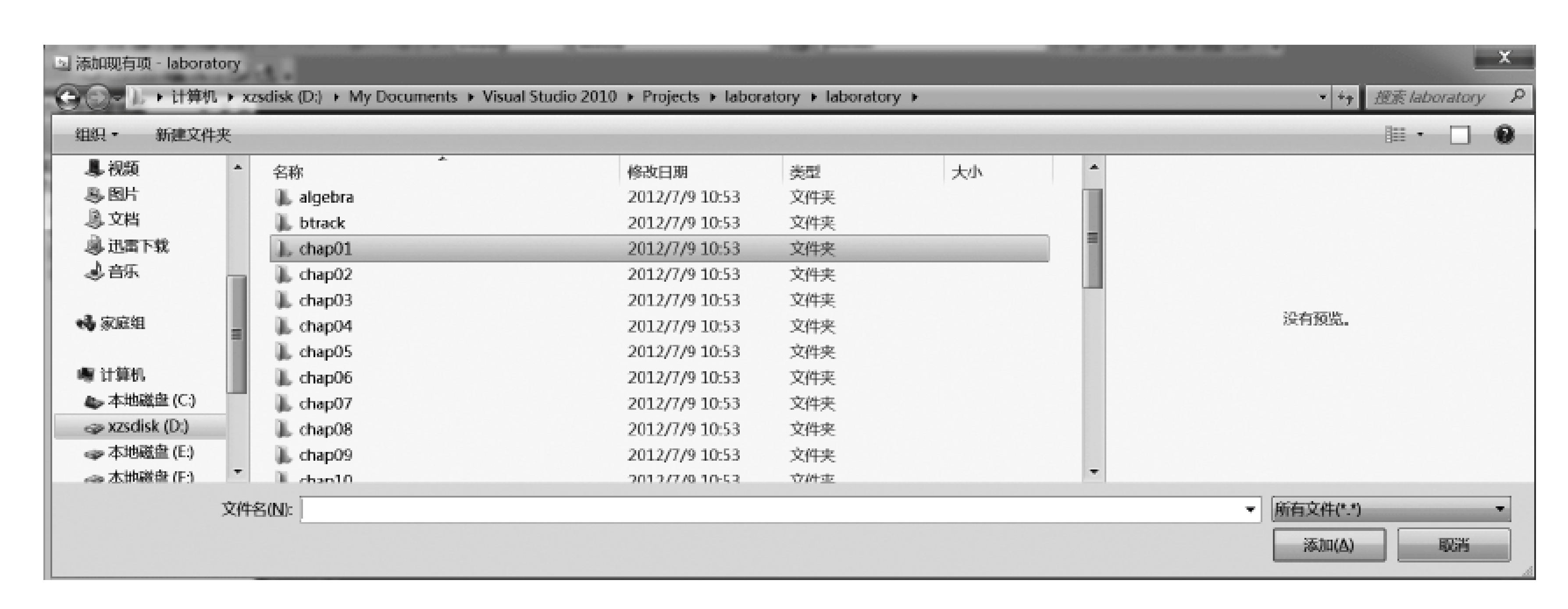


图 12-6　”添加现有项“对话框

双击文件夹图标 chap01，在打开的文件夹中双击文件夹图标 SearchEnging，对话框如图 12-7 所示。在打开的文件夹中选择源文件 searchengine. c，单击“添加”按钮。

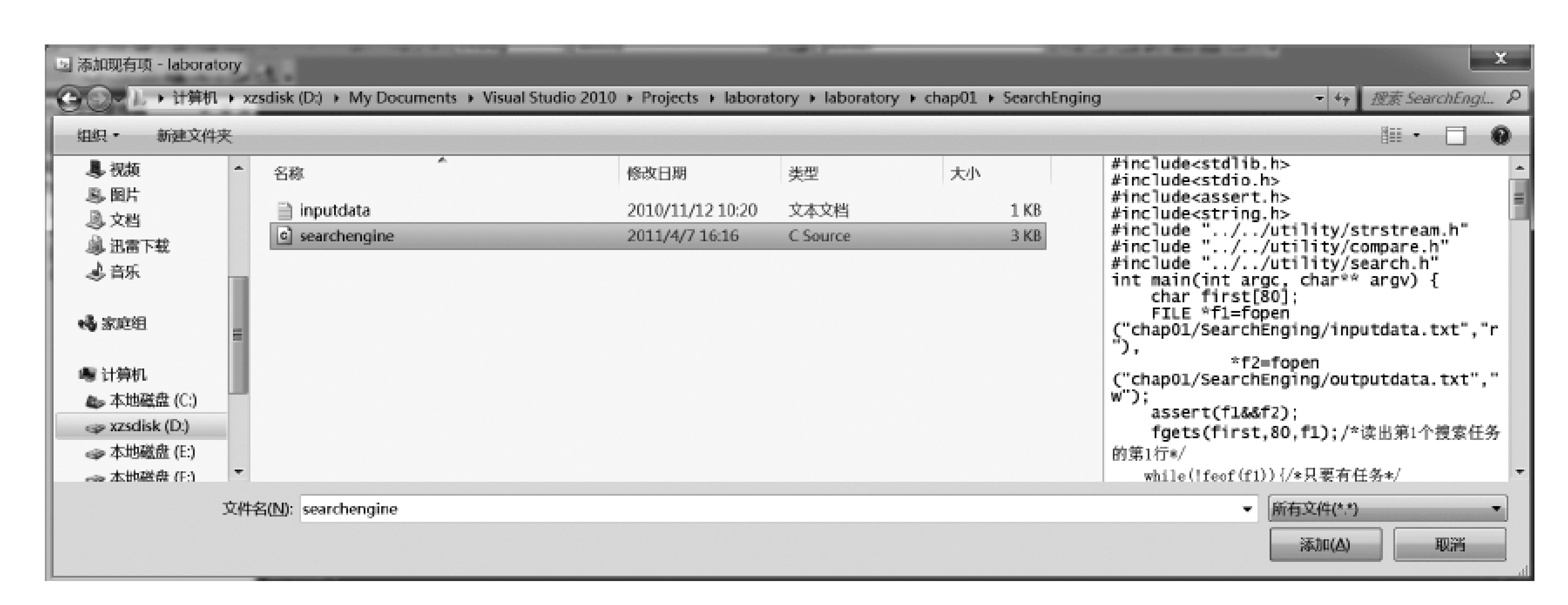


图 12-7　选择需添加的程序源文件

双击 IDE 左窗格中文件 searchengine. c 图标，在编辑区域内打开该文件。此时，IDE 窗口形象如图 12-8 所示。

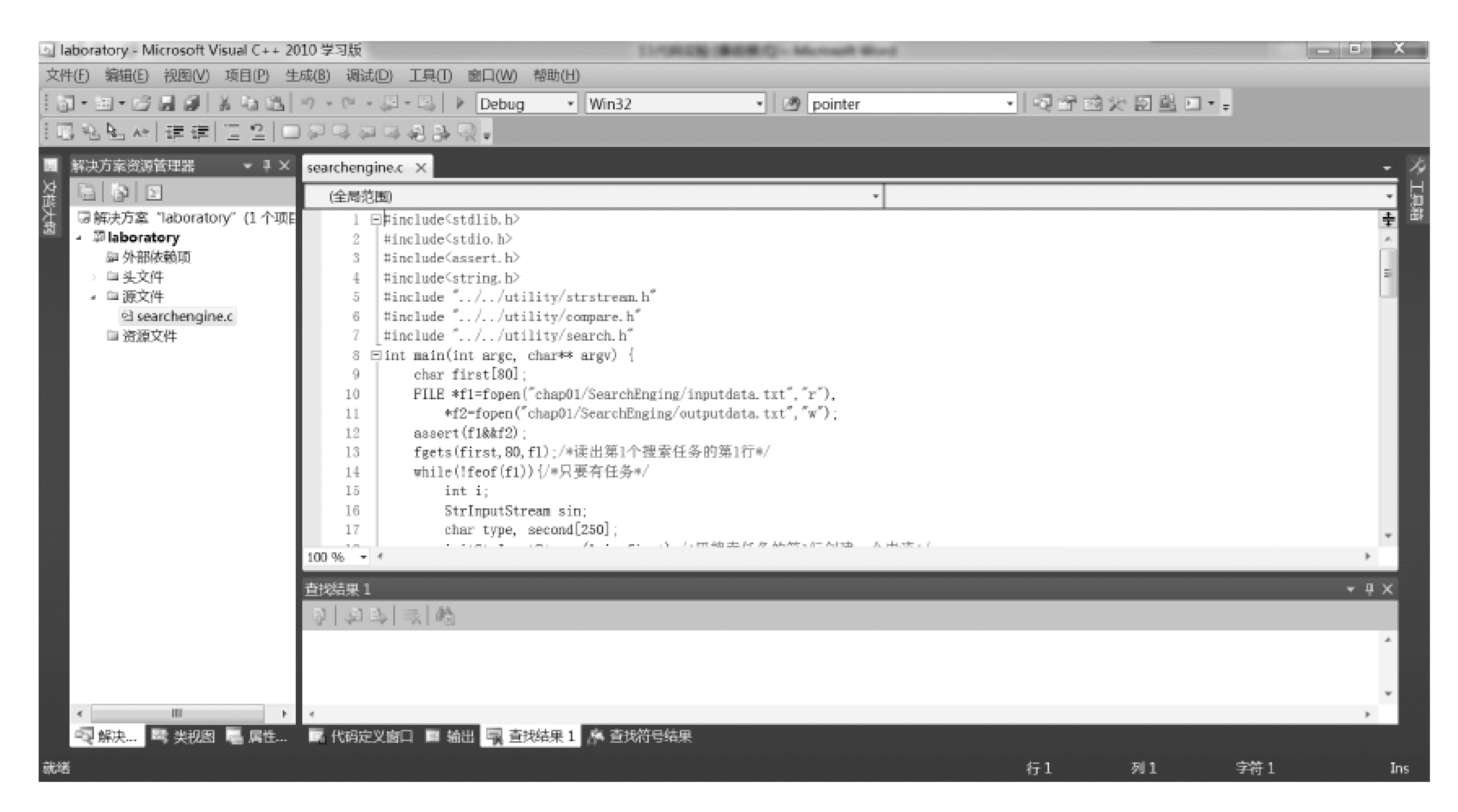


图 12-8　在 IDE 编辑区中打开源文件

(3) 重复第(2)步。继续在项目中添加现有项 utility/strstream.c、utility/compare.c 和 utility/search.h。

12.3.2 调试程序

程序文件加载完毕后,就可以对程序进行调试了。

(1) 在 IDE 中单击"生成"→"重新生成 laboratory",编译、链接程序。

(2) 在程序中合适的地方设置断点(单击行号前区域,断点显示为一红色小球,再次单击将取消断点),如图 12-9 所示。

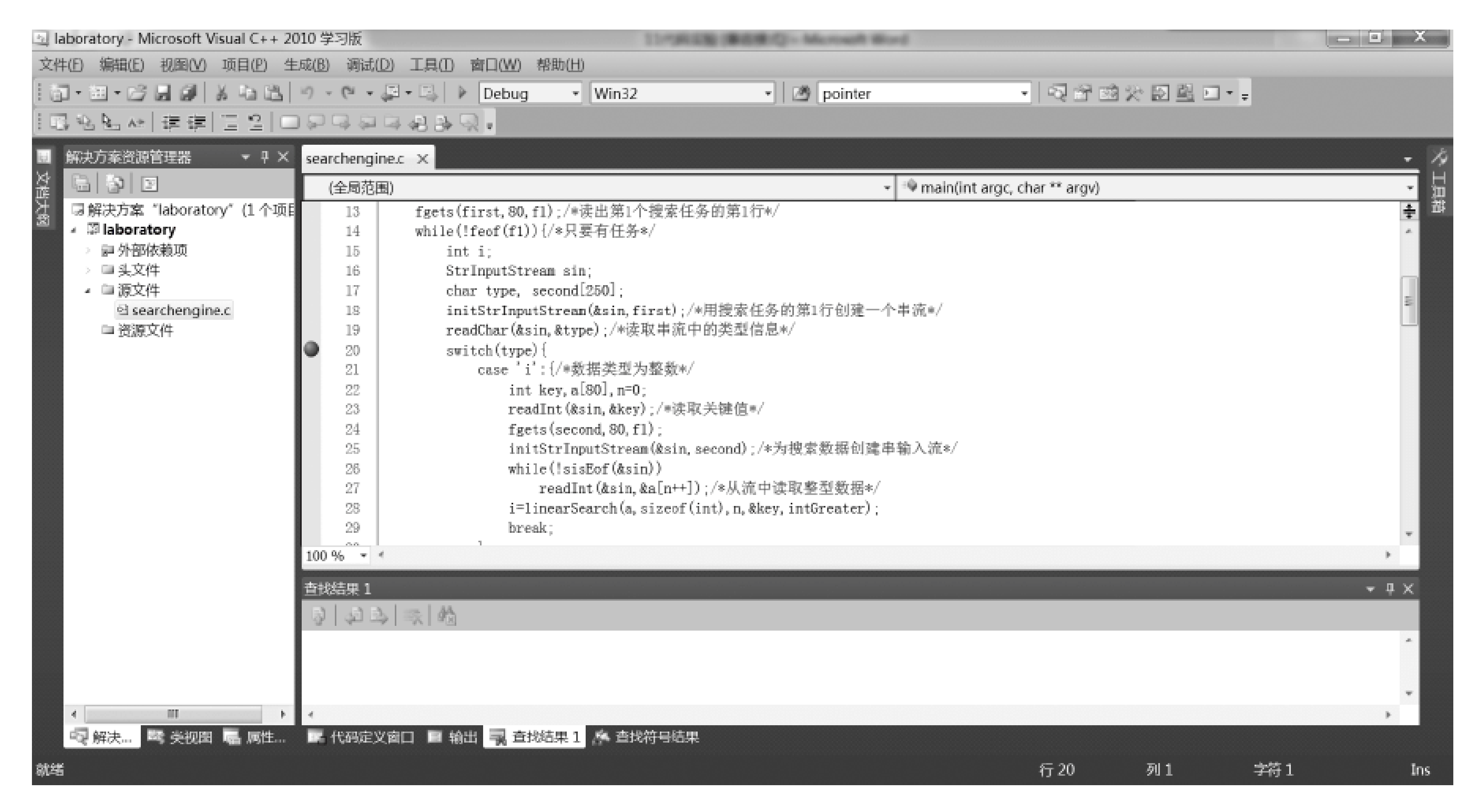


图 12-9 在程序中设置断点

(3) 菜单"调试"→"启动调试"。程序执行到第 1 个断点(见图 12-10)处即可逐过程或逐语句地调试程序了。可以在下窗格中观察各个局部变量的当前值。

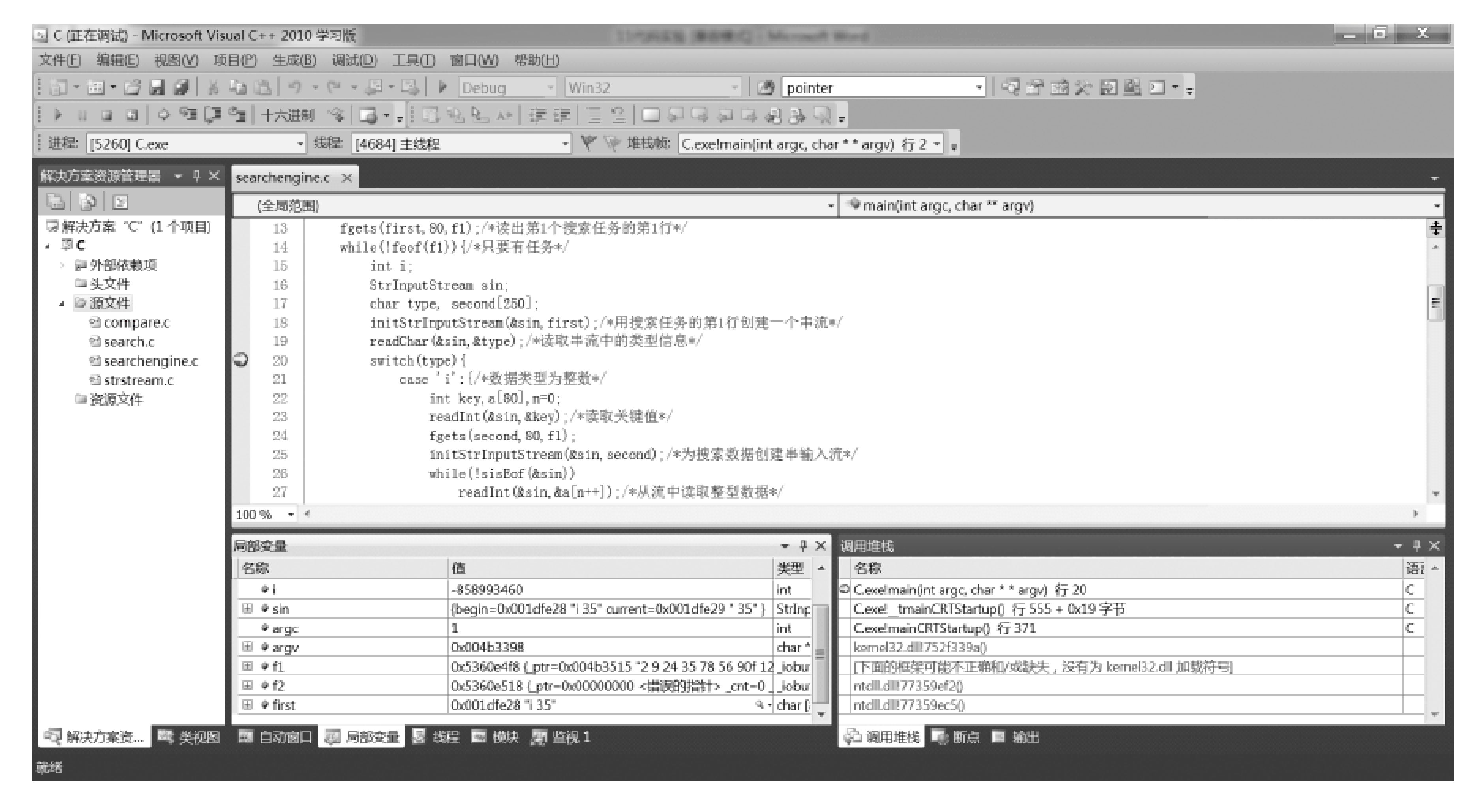


图 12-10 设置断点调试程序

12.3.3　各应用问题加载文件清单

本书从第 1～10 章一共有 45 个应用问题，对每个问题均给出了完整的 C 程序。读者可按 11.3.2 节的步骤方法加载运行程序所需的文件，调试运行之。需要注意的是，在加载一个新的程序所需文件前，如果项目中含有其他程序的文件，应该在右窗格中的"源文件"逻辑目录下右击每一个文件，并在弹出式菜单中点选"从项目中排除"，加以清理（千万不要选"从项目中移除"，那样会导致文件从磁盘中彻底删除）。本节列出这 45 个应用问题程序加载文件清单，方便读者实验。

第 1 章

1. 搜索引擎问题

chap01/SearchEnging/searchengine.c
utility/compare.c
utility/search.h
utility/strstream.c

2. Joseph 问题

chap01/Joseph/Joseph.c

3. Esecape 问题

chap01/Esecape/Escape

4. Counting Quadrangles 问题

chap01/Counting Quadrangles/CountingQuadrangles.c

第 2 章

1. 数据规范问题(链表版本)

chap02/normalize/normalize.c
datastructure/list.c
utility/compare.c
utility/strstream.c

2. Eventually periodic sequence 问题

chap02/Eventually periodic sequence/Eventuallyperiodicsequence.c
datastructure/stack.c
datastructure/list.c
utility/compare.c
utility/strstream.c
utility/search.c

3. 像素转换问题

chap02/pixeltransform/pixeltransform. c
datastructure/queue. c
datastructure/list. c

4. What Fix Notation 问题

chap02/What Fix Notation/WhatFixNotation. c
datastructure/binarytree. c
utility/strstream. c
utility/output. c

5. 数据规范问题(红黑树树版本)

chap02/normalize/normalize1. c
datastructure/redblacktree. c
utility/compare. c
utility/strstream. c

6. Crazy Search 问题

chap02/Crazy Search/CrazySearch. c
datastructure/hashtable. c
datastructure/list. c
utility/compare. c

第 3 章

1. Tour de France 问题

chap03/Tour De France/TourdeFrance. c
utility/randomn. c
utility/compare. c
utility/partition. c
utility/quicksort. c
utility/select. c
utility/swap. c

2. GetTheInversion 问题

chap03/GetTheInversion/GetTheInversion. c

3. Department 问题

chap03/Department/Department. c
utility/heap. c
utility/swap. c
datastructure/list. c
datastructure/pqueue. c

第 4 章

1. Complete the sequence 问题

chap04/Complete the sequence/Completethesequence. c
algebra/lup. c
algebra/matrix. c
algebra/poly. c
algebra/rational. c
utility/swap. c

2. Equivalent Polynomial 问题

chap04/Equivalent Polynomial/EquivalentPolynomial. c
algebra/poly. c

3. Rational Approximation 问题

chap04/Rational Approximation/RationalApproximation. c
algebra/lup. c
algebra/matrix. c
algebra/poly. c
algebra/rational. c
utility/swap. c

第 5 章

1. PiPe 问题

chap05/pipe/pipe. c
geometry/point. c

2. Smallest Bounding Rectangle 问题

chap05/Smallest Bounding Rectangle/SmallestBoundingRectangle. c
chap05/overlaprectangle. c
datastructure/list. c
datastructure/stack. c
geometry/convexhull. c
geometry/point. c
utility/most. c
utility/partition. c
utility/quicksort. c
utility/swap. c

3. Texas Trip 问题

chap05/TexasTrip/TexasTrip. c
chap05/overlaprectangle. c
datastructure/list. c
datastructure/stack. c

geometry/convexhull. c
geometry/point. c
utility/most. c
utility/partition. c
utility/quicksort. c
utility/swap. c

第 6 章

1. The X Problem 问题

chap06/The X Problem/TheXProblem. c
numbertheory/bigint. c
numbertheory/valuecalc. c
datastructure/list. c

2. The Hardest Problem Ever 问题

chap06/The Hardest Problem Ever/TheHardestProblemEver. c

3. Jugs 问题

chap06/Jugs/Jugs. c
numbertheory/bigint. c
numbertheory/valuecalc. c
numbertheory/gcd. c
datastructure/list. c

4. 青蛙约会问题

chap06/青蛙约会/青蛙约会. c
numbertheory/bigint. c
numbertheory/valuecalc. c
numbertheory/gcd. c
numbertheory/lequation. c
datastructure/list. c

5. Smith Numbers 问题

chap06/Smith Numbers/SmithNumbers. c
numbertheory/bigint. c
numbertheory/valuecalc. c
numbertheory/exponent. c
numbertheory/factor. c
numbertheory/gcd. c
numbertheory/primetest. c
datastructure/hstb. c
datastructure/hstb1. c
datastructure/list. c
datastructure/redblacktree. c

utility/compare. c
utility/random. c

第 7 章

1. Queens in peaceful positions 问题

chap07/Queens in peaceful positions/Queensinpeacefulpositions. c
btrack/combineproblem. c
btrack/permutetree. c
datastructure/list. c
datastructure/stack. c
utility/swap. c

2. Color the Map 问题

chap07/Color the Map/ColortheMap. c
btrack/backtrackiter. c
btrack/combineproblem. c
btrack/mcolor. c
datastructure/list. c
geometry/point. c

3. Divisibility 问题

chap07/Divisibility/Divisibility. c
btrack/combineproblem. c
btrack/subsettree. c
datastructure/list. c

4. Calculate A Route 问题

chap07/Calculate A Route/CalculateARoute. c
btrack/combineproblem. c
btrack/permutetree. c
datastructure/list. c
datastructure/stack. c
utility/swap. c

5. Graph Coloring 问题

chap07/Graph Coloring/GraphColoring. c
btrack/combineproblem. c
btrack/subsettree. c
datastructure/list. c

6. Communication System 问题

chap07/Communication System/CommunicationSystem. c
btrack/backtrackiter. c
btrack/combineproblem. c

datastructure/list. c

第 8 章

1. Cow Solitaire 问题

chap08/Cow Solitaire/CowSolitaire. c

2. Hero Shoot Eagle 问题

chap08/Hero Shoot Eagle/HeroShootEagle. c
dprog/lcs. c
datastructure/list. c
utility/compare. c
utility/partition. c
utility/quicksort. c
utility/strstream. c
utility/swap. c

3. Human Gene Functions 问题

chap08/Human Gene Functions/HumanGeneFunctions. c

4. Happy Travel 问题

chap08/Happy Travel/HappyTravel. c
dprog/knapsack. c

5. The cicerone's abacus 问题

chap08/The cicerone's abacus/The cicerone's abacus. c
dprog/knapsack. c
datastructure/pair. c

6. Bronze Cow Party 问题

chap08/Bronze Cow Party/BronzeCowParty. c
dprog/floyd. c
datastructure/pair. c
utility/compare. c
utility/partition. c
utility/select. c
utility/swap. c
utility /randomn. c

第 9 章

1. Radar Installations 问题

chap09/Radar Installations/RadarInstallations. c
greedy/actsel. c

2. Variable Radix Huffman Encoding 问题

chap09/Variable Radix Huffman Encoding /VariableRadixHuffmanEncoding. c
greedy/huffman. c
datastructure/pqueue. c
utility/heap. c
utility/swap. c

3. Arctic Network 问题

chap09/Arctic Network/ArcticNetwork. c
greedy/prim. c
datastructure/pqueue. c
datastructure/pair. c
utility/heap. c
utility/swap. c
utility/compare. c

4. Pipe Laying Problem 问题

chap09/Pipe Laying Problem/PipeLayingProblem. c
greedy/dijkstra. c
datastructure/pqueue. c
datastructure/pair. c
gepmetry/point. c
utility/heap. c
utility/swap. c
utility/compare. c

第 10 章

1. Sorting It All Out 问题

chap10/Sorting It All Out/sortingitallout. c
datastructure/list. c
datastructure/pair. c
datastructure/stack. c
graph/graph. c
graph/tpsort. c

2. Quote Calling Circles 问题

chap10/Quote Calling Circles/Quotecallingcircles. c
datastructure/list. c
datastructure/pair. c
datastructure/stack. c
graph/graph. c
graph/scc. c

3. Bonnie and Clyde 问题

chap10/Bonnie and Clyde/BonnieandClyde. c
datastructure/list. c
datastructure/pair. c
datastructure/stack. c
graph/graph. c
graph/dfs. c
graph/articpoint. c

4. Risk 问题

chap10/Risk/Risk. c
datastructure/list. c
datastructure/pair. c
datastructure/queue. c
graph/graph. c
graph/bfs. c

5. Internet Bandwidth 问题

chap10/Internet Bandwidth/InternetBandwidth. c
datastructure/list. c
datastructure/pair. c
datastructure/queue. c
graph/graph. c
graph/bfs. c
graph/edmondsKarp. c

第 11 章

1. DNA Laboratory 问题

chap11/DNA Laboratory/DNALaboratory. c
datastructure/list. c
utility/listpatition. c
utility/listquiksort. c
utility/random. c
utility/swap. c

2. Palindrome 问题

chap11/Palindrome/Palindrome. c
utility/textmatch. c

3. String Matching 问题

chap11/String Matching/StringMatching. c
algebra/rational. c

12.4　函数库的扩展

可以用两种方式向函数库中增加函数，其一是向已有的源文件中添加函数定义代码，在相应的的头文件中添加函数的原型声明。其二是添加独立存储的新源文件及相应的头文件。本节以第 1 章中算法 1-2 描述的二分查找算法的实现为例，说明函数库扩展的这两种方法。

12.4.1　向已有的源文件中添加新函数

在第 1 章的节 1.1 中我们知道二分查找算法适用于在排好序的序列中查找等于指定值的元素。我们把算法 1-2 的伪代码过程再现于下。

```
BINARY-SEARCH(A, x)
p←1, r←length[A]
while p≤r
    do q←⌊(p+r)/2⌋
       if A[q]=x
          then return q
      if A[q]<x
         then p←q+1
         else r←q−1
return −1
```

其中，参数 A 表示一个有序序列，x 表示待查的关键值。

可以在 C 语言中将此算法实现为一个对存储为数组的有序序列 a 查找特定关键值 x 的函数。

```
int binSearch(void *a, int size, int n, void *x, int(*comp)(void*, void*)){
    int p=0,r=n-1,q,c;
    while(p<=r){
        q=(p+r)/2;
        c=comp(x, (char*)a+q*size);
        if(c==0)
            return q;
        if(c>0)
            p=q+1;
        else
            r=q-1;
    }
    return -1;
}
```

在实验项目中加载 Utility 文件夹下的源文件 search.c。将上述函数 binSearch 的定义代码添加在该文件尾部。

在实验项目中加载 Utility 文件夹下的头文件 search. h。将函数 binSearch 的原型声明

```
int binSearch(void *a, int size, int n, void *x, int(*comp)(void*, void*));
```

添加进去。

然后重新生成项目。若在应用中需要调用此函数,可以在适当的位置添加指令"#include "…utility/search. h"",并在项目中加载源文件 search. c。

12.4.2 创建新的源文件

可以用以下代码对链表实现算法 1-2。

```
ListNode *listBinSearch(LinkedList *L, void *x){
    int p=0,r=L->n-1,q,c;
    ListNode *P=L->nil->next, *R=L->nil->prev, *Q;
    while(p<=r){
        q=(p+r)/2;
        Q=advance(P,q);
        c=L->comp(x, Q->key);
        if(c==0)
            return Q;
        if(c>0){
            P=Q->next;
            p=q+1;
        }else{
            R=Q->prev;
            r=q-1;
        }
    }
    return NULL;
}
```

由于对链表的先行查找已经实现为 list. c 中的一个函数 listSearch,可以把上述函数存储为独立的源文件,便于代码重用。为此,在项目中向文件夹 Utility 添加新的源文件 listbinsearch. c,将上述代码录入其中。在同一文件夹中添加新的头文件 listbinsearch. h,将下列与变异指令和函数 listBinSearch 的原型声明

```
#include "../datastructure/list. h"
    ListNode *listBinSearch(LinkedList *L, void *x);
```

添加进去,重新生成项目。在需要调用该函数的应用项目中合适的地方添加指令"#include "…utility/listbinsearch. h"",并在项目中加载 Utility 文件夹中的源文件 listbinsearch. c。

参 考 文 献

[1] Ronald Graham，Donald Knuth ，Oren Patashnik. 具体数学：计算机科学基础(第 2 版)[M]. 北京：机械工业出版社，2002.

[2] 朱洪，等. 离散数学教程[M]. 上海：上海科学技术文献出版社，1996.

[3] Thomas H. Cormen，Chales E. Leiserson，Ronald L. Rivest，Clofford Stein. 算法导论(第二版)[M]. 北京：高等教育出版社，2002.

[4] Alfred V. Aho，John E. Hopcroft，Jeffrey D. Ullman 著. 计算机算法的设计与分析[M]. 黄林鹏，王德俊，张仕 译. 北京：机械工业出版社，2007.

[5] Anany Levitin 著. 算法设计与分析基础[M]. 潘彦译. 北京：清华大学出版社，2003.

[6] 严蔚敏，吴伟民. 数据结构(C 语言版)[M]. 北京：清华大学出版社，1997.

[7] Brian W. Kernighan，Dennis M Ritchie 著. C 程序设计语言[M]. 徐宝文，李志译. 北京：机械工业出版社，2007.

[8] Peter Van Der Linden 著. C 专家编程[M]. 徐波译. 北京：人民邮电出版社，2008.